Nam Nguyen

Development & deployment of a web server as an executable with Node.js, Express.js and Vercel/pkg

Metropolia University of Applied Sciences

Bachelor of Engineering

Software Engineering

Bachelor's Thesis

18 January 2022

Author	Nam Nguyen
Title	Development & deployment of a web server as an executable with Node.js, Express.js and Vercel/pkg
Number of pages	58 pages
Date	18 January 2022
Degree	Bachelor of Engineering
Degree programme	Information Technology
Professional Major	Software Engineering
Instructors	Janne Salonen, Head of Department ICT

The purpose of the project is to implement an application that would serve as a license web server which could be deployed both as a cloud solution and an on-premises setting for Windows operating system. Keeping the requirements of the client users in mind, the solution aims to offer the least complex approach for setting up the application on the end-user side.

The technology stack chosen to implement the project has been Node.js/TypeScript, Express, Sequelize, SQLite and Vercel/pkg among other libraries. The reason behind the selection of the stack lies in the fact that JavaScript/Node.js is a powerful language which has a versatile usage in numerous ways.

The thesis work would be the technical documentation of how the project is planned, implemented and what the technical details related to the libraries and tools in use are. It serves also as a documentation for the project as a proof of concept how similar implementation can be done with the selected technology for the same requirements.

Keywords	Node.js, TypeScript, Express, Sequelize, SQLite, Vercel/pkg

Table of Contents

1 Introduction *5*

2 Theoretical background *6*

2.1 Node.js **6**

2.2 Typescript **8**

2.3 Express.js **10**

2.4 SQLite **11**

2.5 SequelizeORM **13**

2.6 Vercel/pkg **14**

3 Implementation *18*

3.1 Project objectives **18**

3.2 Project architecture **20**

3.3 Project structure **21**

3.4 Project implementation **25**

3.4.1 Express server setup 26

3.4.2 Database schema and configuration 33

3.4.3 Command line tool 37

3.5 Project packaging **39**

3.5.1 Packaging process 39

3.5.2 Packaging script 43

3.5.3 Build commands and custom options 45

3.5.4 Continuous integration pipeline with GitHub Actions 47

4 Results and discussions *51*

5 Conclusions *55*

6 References *56*

List of Abbreviations

ACID:	A set of properties of database transactions. Stands for Atomicity, Consistency, Isolation, Durability
API:	Application Programming Interface
CLI:	Command line interface
CRUD:	Basic operations of persistent storage. Stands for Create, Read, Update, Delete
DBMS:	Database management system. Software for maintaining, querying, and updating data and metadata in a database.
ECMA:	JavaScript standards defined by Ecma International
HTML:	HyperText Markup Language
HTTP:	Hypertext Transfer Protocol
IDE:	Integrated Development Environment
I/O:	Input/Output
IIFE:	Immediately Invoked Function Expression
ORM:	Object-relational mapping. The set of rules for mapping objects in a programming language to records in a relational database, and vice versa.
SQL:	Structured Query Language
TCP/IP:	Transmission Control Protocol/Internet Protocol

1 Introduction

Nowadays it is common approach to utilize cloud infrastructure for application deployment and management. By migrating to the cloud, the developers can focus less on managing or administrating work and invest more resources into practical development of applications and features that would be value-added.

However, there are cases in which the end user would need a technical solution that would work in a closed environment with no connection to open internet. This is the primary driving force behind the project, in that the end user would like to have a working server that serves as a licensing web server dedicated to the client tools. The application server should have its own database, in which it can set up by itself without the end user having to operate a database server to support the application. No external communication from the licensing server is allowed.

The environment which the license server should operate in is Windows Server, and that the requirements indicate the least effort in setting up and maintaining the application server as possible. In fact, the license server could be just plug-and-play style, in which the end user could simply start the application without needing to install any library or containerization service like Node.js, Docker etc.

The thesis work is done to serve as a technical documentation for the project, which is exploratory in nature, and that the project itself is a proof of concept how it is feasible technically. At the end, the outcome of the project is a minimum viable product that the end user can start testing, and that it should satisfy all the requirements agreed in prior to the start of the implementation. The thesis would also be a valuable reference for any future need for the same implementation approach with the same requirements.

2 Theoretical background

This chapter aims at providing theoretical explanation of the programming language, the web framework, and the packaging tool to be used within the project.

2.1 Node.js

Node.js in essence is an open-source server-side JavaScript runtime environment which is capable of cross-platform operation. It was built on top of Chrome's V8 engine, which enables it to run JavaScript code outside of the browser. [1]

Node.js has a versatile usage, and is suitable for general-purpose programming, hence software developers can utilize Node.js to create almost any kind of program, ranging from web servers, games, scripts, CLI application and so on. [1]

Created by Ryan Dahl in 2009, Node.js was purposefully written to handle and address the issues which the most widely used web server at that time, Apache HTTP Server, was encountering. On the first European JSConf in November 2009, Ryan Dahl demonstrated the use of Node.js project, which in later phases was also championed by other companies like Microsoft and Mozilla. [3]

Node.js has an architecture which is of single-threaded and event-driven characteristics. This event-driven notion is also associated with another equally popular term, which is 'event loop'. Due to these characteristics, Node.js is lightweight, highly scalable, and high performant. It employs asynchronous I/O operations which makes its working approach non-blocking in comparison to traditional multi-threaded processing model. [4]

Figure 1 shows the architecture of Node.js, and how event loop which is at the core of the platform can be understood.

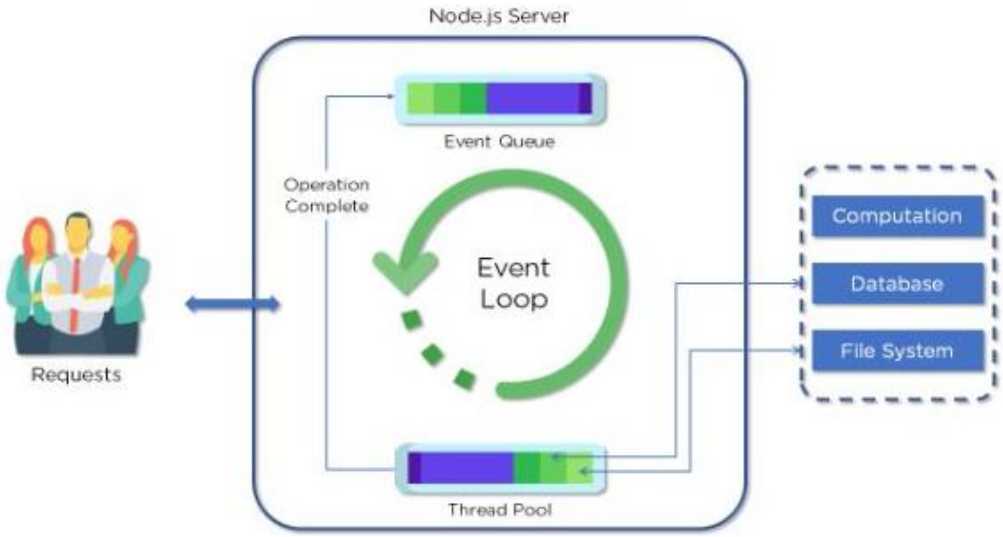


Figure 1. Node.js architecture with event loop

Figure 1 depicts how requests are processed by a Node.js server. Clients will interact with Node.js web server by sending requests to it, which it in turn will place those requests to the event queue. The type of request can vary, being either of blocking or non-blocking nature as it depends on the complexity of the task demanded by the request. [4]

From inside of event queue, the Node.js application server will dispatch one by one each request to the event loop. Event loop will be responsible for handling primarily simple requests which are non-blocking such as I/O polling. After that, event loop returns the response back to the client who is the owner of the request. [4]

For such requests that demand heavy computational power, they are considered as blocking operations. Thus, event loop will send those requests to the thread pool, from which an individual thread will be assigned for the handling of each of mentioned heavy requests. This type of request occasionally involves read/write operations to the database system, heavy computations for encryption and so on. When the thread has finished the assigned task, it returns the response from the thread pool back to event loop, and from which responses are subsequently passed back to the clients. [4]

Node.js has been on the rise in terms of usage since its inception in 2009. According to a StackOverflow survey in 2019, nearly 50% of respondents out of 58,543 interviewees gave a positive opinion on Node.js. [5] Some of the biggest companies in the world are using Node.js in a production setting, for example LinkedIn, eBay, Netflix and many more. [6]

2.2 Typescript

Typescript is a programming language by itself which was crafted by Microsoft back in early 2010 and spearheaded by Danish software engineer Anders Hejlsberg. Shortly afterwards, Typescript was transitioned to be open sourced in 2012. In essence, Typescript is a superset of JavaScript or, in other words, JavaScript that comes with types. [7]

Because of how Typescript was built on top of JavaScript, the two programming languages share moderate similarities in terms of syntax. Nevertheless, a significant number of differences exist between the two languages which is worth mentioning.

Table 1 illustrates how Typescript and JavaScript differ from each other in many aspects. Selected aspects for comparison are considered major as they clearly play a role in distinguishing the two languages.

Table 1. Major differences between TypeScript and JavaScript

Aspect	Typescript	JavaScript
Annotation and typing	Strongly typed language	Weakly/dynamically typed language
Data Binding	Possible with types and interfaces	Not possible
File extension used	.ts and .tsx	.js
Compiler requirement	Yes	No

From table 1, the major differences between the two languages are listed. They are respectively annotation and typing, how data binding is enabled, file extension in use, requirement for compilation and use case suitability.

It is conceived that TypeScript is a strongly typed language, which belongs to a group of other languages that do type checking at compilation time, e.g., Java, Haskell, C#, Scala, and some others. JavaScript, on the other hand, is itself a dynamically typed programming language that is in the same category with known languages such as Ruby, PHP, Python, Perl etc. [8] Basically, JavaScript does not require users to specify the types of data being used during variable declarations. It is the opposite for TypeScript where data should always be annotated with types, which can happen through explicit user declaration or implicit inference. [8]

TypeScript is equipped with core notions of types and interfaces with which users can annotate the type of data to be used. This, from JavaScript's point of view, is impossible as the language does not support type annotation. The resulting consequence is that the developer is solely responsible for the correctness of how variable types are declared and used. In contrast, TypeScript, with the support from developers' IDE, helps developers to spot out inaccurate use of variables during development time. [9]

TypeScript uses .ts or .tsx as its file extension, while JavaScript is associated with .js files. This is the clearest indication which language is being used for a particular project or file. [9]

An important aspect related to extension file is the compilation requirement for executing TypeScript and JavaScript files. Although TypeScript is the superset of JavaScript, browsers cannot execute TypeScript codes directly. All TypeScript files need to go through a process called trans-compilation, in which TypeScript Checker compiler or Babel compiler will compile TypeScript code into plain JavaScript code. The result of the compilation process means that at runtime, all codes which were written in TypeScript would then be plain JavaScript files.

Notably, JavaScript does not need to compile its code to run. On the browser, developers can include .js scripts in HTML files with <src> tag for instance. From server-side perspective, the situation is the same where Node.js runs JavaScript but not directly TypeScript. [9] [10]

TypeScript has considerable benefits over plain JavaScript when a developer must decide on a language to use for a project. As previously mentioned, TypeScript offers native implementation to adopt typing of the data, and that software developers can have better code structure and better error discovery during development. This is of paramount importance where application failure due to incorrect usage of data types can already be prevented before it executes. [9]

TypeScript, through configuration, is capable of backward compatibility, in which users can define a specific ECMAScript version of the target JavaScript code, e.g., ES6, ESNext or many others. This supports using TypeScript on older browsers whose version is limited to a particular JavaScript ECMAScript version. [9] Nonetheless, for small projects, using TypeScript will result in some overhead because of compilation requirement in prior to execution. TypeScript development environment must also be configured in advance, whilst JavaScript code can simply be written and executed without needing for heavy setup. [9]

2.3 Express.js

Express is an open-sourced, minimalistic, and flexible web application framework hosted within Node.js runtime environment. It comes equipped with a versatile set of functionalities for building and designing web and mobile applications. Express is one of the most popular web framework for Node.js as of date, which was written in JavaScript and is comfortable for programmers to learn and utilize. [11] [12]

The use case for Express.js is highly diverse. Developers can build single page applications or hybrid web applications with Express.js. Because Express.js is a

Node.js framework, applications built with Express.js can be deployed to any target environment where Node.js can operate. [12]

There are numerous benefits of using Express.js as a web framework. The framework is lightweight, which enables efficient API routing. Express.js is empowered by middleware modules, which executes after the reception of requests from server and before the server returns responses back to clients. In fact, an Express application consists of a series of middleware function calls. With middleware module, developers can, among other aspects, perform optimization on output before client-side delivery, improve security aspects of the application, or interact with a request object as many times needed. [13]

Due to the framework's nature being JavaScript under the hood, the learning curve for Express.js is comfortable for developers. On top of that, the open-source community behind the framework offers profound support for any issues or new ideas, which pushes the durability and quality of the framework even further. [13]

2.4 SQLite

SQLite is a software package that provides relational database management system. This means SQLite shares some similarities with other popular RDBMS products such as Microsoft's SQL Server, IBM's DB2 from the commercial sphere and MySQL or PostgreSQL representing open-source community. [14]

SQLite's name has not come without confusion to those who did not have prior knowledge of it as to what it means. In fact, the 'lite' part in SQLite does not impose any implication on its capability. The correct understanding should refer to its lightweight setup, low administrative overhead, and low consumption of resource. [14]

The most discernible features associated with SQLite are self-containment, serverlessness, zero-configuration, cross-platform operation, and transactional

data handling. Self-containment means SQLite does not need heavy support from operating system or external library to run. SQLite, as a library, contains the database system as a whole entity which integrates straight into host application. This essentially enables SQLite usage on embedded devices like handheld devices (iOS or Android). [14] [15]

SQLite's serverlessness can be understood that the library does not require a database server process to operate, which is known as client-server architecture. While other RDBMS systems like MySQL or PostgreSQL need a running server to request through TCP/IP protocol for sending and receiving data, SQLite, on the contrary, connects with its database storage file. This file resides on the host system where the application using SQLite is. The read/write operation then occurs directly back and forth from this storage file. [14] [15]

Having zero-configuration means that SQLite does not require installation before using. Being serverless in nature results in no server setup, configuration, or similar prerequisites. In addition, SQLite does not need any configuration files to start with. [14] [15] When SQLite runs in cross-platform environments, the whole database instance operates within one storage file, and such file can be used across different operating systems. [14]

SQLite is also capable of offering fully transactional operations. All transactions executed by SQLite library are ACID-compliant, allowing reliable access from different threads or processes. During unexpected crash or failure, a transaction performed by SQLite either completes successfully or not at all. [15]

SQLite's use case is significantly suitable for situations in which complex setup of dedicated database server is not feasible or necessary. It is in use for testing, prototyping, development and many more. A noticeable point worth mentioning is that SQLite is not considered as replacement to full-fledged RDBMS systems, but a complement for them. SQLite fits in deployment environment where simplicity and ease of use are preferred over capacity, concurrency but heavy

setup associated with other RDBMS systems that use client-server architecture. [15]

2.5 SequelizeORM

Sequelize is a promise-based Object Relational Mapper (ORM) for Node.js that supports a multitude of different SQL databases such as MySQL, MariaDB, PostgreSQL, SQLite and MSSQL. The ORM library offers solid transaction support, relations, eager and lazy loading, read replication among many other features. [16]

Object Relational Mapping can be understood as a technique providing a layer of abstraction over data usage in a relational database. ORM will map or create mapping that would convert data from relational database to specific objects within the target programming language. The purpose of ORM libraries like Sequelize is to link details between data sources that cannot coexist due to mismatches. [16]

Benefits from using Sequelize library as an ORM are noticeable. Using Sequelize would require much less effort and time during development, as programmers no longer need to write raw SQL queries to perform transactions to databases.

This reduces the margin of errors related to typos or incorrect statements, especially when different relational database dialects might have their specific query syntax. Developers would only be required to perform initial setup for Sequelize with corresponding SQL dialect, connection, privilege, and so on without actual need of knowing in detail about underlying database dialect. [16]

Alongside aforementioned features that Sequelize offers, the library additionally extends support for other functionalities such as model hooks, transactional support, database migration, model validation, seeding, scope etc. All this help abstracting complex CRUD queries and enables true declarative code in an application development. [16]

Many companies are reportedly using Sequelize in production, some of which can be named as Goopy, airCloset, BaseDash, OWA, WalmartLabs etc. Weekly downloads of the library tops at around 1 million, and it has a popularity in top 5% for ORM alternatives and currently in use in over 3,600 projects. [17] [18]

2.6 Vercel/pkg

Pkg, or Vercel/pkg, is an open-source command line utility tool that enables the packaging of Node.js projects into executables that can run even in environments where Node.js is not present. [19] Some other similar tools which can be considered as alternatives to pkg are nexe, ncc etc.

According to the official pkg website, the utility covers a considerable of different use cases. Some of which can be listed as creation of application without sources, cross-platform compilation, single file execution/deployment, testing of new Node version, execution in non-Node.js environment etc. [19]

Pkg functions in a way that it will firstly fetch the base Node.js binary needed from Github releases depending on the request node version, arch, and operating system. Besides direct binary fetching, pkg can also look inside cache folder on the machine building the project for the necessary Node.js binary image. The utility will, as a next step, traverses through the source code of the application and dependencies to detect what to include into the build. From there, pkg runs source JavaScript code through V8 compiler and eventually produces a V8 snapshot from the base Node.js binary, which then stores the project’s code. [20]

The utility can be used either programmatically or as command line interface. Regardless, pkg equips the users with a myriad of different options to customize the build process. They range from target operating system selection, output directory, executable naming, compression, and so on.

As mentioned, pkg can use different Node version as base to build an executable for target operating systems being Windows, macOS, Linux, Alpine etc. with

chosen arch. Additionally, the tool offers the capability to include scripts and static assets into the final build, which makes it even more portable. During packaging process, pkg attempts to include every necessary dependency which an application would need. [19]

However, there are cases where reference to static assets or non-JavaScript files like images, views, stylesheets etc. and non-literal argument to require calls is needed. Non-literal argument to require calls can be understood as when the developers are making require function calls to his/her own JavaScript script files, and not a dependency or Node module, in other parts of the application. These use cases will not be handled automatically by the pkg library, but the developer can explicitly declare in a manual manner which files should be included in the final build. [19]

Pkg utility does support compression for minimizing the size of the output executable. Two different compression means are supported, either with Brotli or Gzip. This is said to be reducing the size of packaged application up to 60%, but it comes with a trade-off that the application may have more delay during start-up sequence. [19]

A noticeable notion about pkg utility is the term snapshot. After packaging process, in which all files are placed into the final executable, the executable itself is called a snapshot, and is considered somewhat a container by itself. During runtime, or when the executable is run, the packaged application is provided access to the snapshot filesystem where all application files reside. [19]

The packaged files would have a prefix of ‘/snapshot/’ in their paths. The executable can get reference to files that are outside of it, or the real filesystem in which it operates, by using the command ‘process.pwd()’ or ‘path.dirname(process.execPath)’ to define the actual location of needed files. This is especially useful in cases where the packaged application needs to collect some json configurations or detect the presence of a text file. For references to files that have been included altogether in the packaging process, values

determined from ‘__filename’, ‘__dirname’ or ‘process.pkg.defaultEntrypoint’ can be used as the base for path calculation. [19]

Table 2 would list how the snapshot filesystem yields different values when the application is packaged with pkg and when the application is executed purely in Node.js runtime.

Table 2. Pkg snapshot filesystem values [19]

Values	With Node	When packaged	Comment
__filename	/project/app.js	/snapshot/project/app.js	
__dirname	/project/	/snapshot/project	
process.pwd()	/project	/deploy	Suppose application is called …
process.execPath	/usr/bin/nodejs	/deploy/app-x64	app-x64 and run in /deploy
process.argv[0]	/usr/bin/nodejs	/deploy/app-x64	

process.argv[1]	/project/app.js	/snapshot/project/app.js	
process.pkg.entrypoint	undefined	/snapshot/project/app.js	
process.pkg.defaultEntrypoint	undefined	/snapshot/project/app.js	
require.main.filename	/project/app.js	/snapshot/project/app.js	

As shown from the table, values for path calculation are different in the case if an application simply runs in Node.js or after it has been packaged by pkg utility. Among these values, process.pkg.entrypoint and process.pkg.defaultEntrypoint are of high importance and will be revisited in later chapters for their use case in determining which setting the application code is executing.

As previously mentioned, pkg utility can be used either via command line interface, which requires prior installation to the system or developers can utilize its programmatic API to perform the same packaging process. An example code which explains the usage can be found from following code snippet. [19]

```
await exec(['app.js', '--target', 'host', '--output', 'app.exe']);
// do something with app.exe, run, test, upload, deploy, etc
```

In later chapter where project implementation is explained, pkg utility will play a crucial role in the packaging process of the developed application and how the

developer has configured the tool to deliver the target output executable will be thoroughly elaborated.

3 Implementation

3.1 Project objectives

The project objective is to provide a solution for a license service in which it would be possible to work as a cloud solution and an on-premises local service. With some of the clients are disinclined to be allowing their internal developers to connect directly to the cloud hosted licensing service, a local and on-site license service serving almost the same functionalities offered by the cloud is of driving force behind this. This is considered as local and on-premises solution since the application will be hosted and exposed only to the intranet of the client network, and no public internet connection is allowed.

By this approach of offering dual deployment method, the application can expand its use case to both cloud-centric customers and closed environment customers. This is also beneficial in a way that only one codebase is needed to fit the different use cases identified.

The following requirements will be achieved by the developed application:

- The client is not required to set up a running database server to support the application. The application can provision by itself a persistence solution by utilizing SQLite.

- The client is not required to set up any containerization daemon e.g., Docker to be able to run the application. In addition, the application can operate in Windows Server operating system where no Node.js installation exists

- No extra installation or dependencies should be required when running the application

- The internal logic of the application can be determined by the different use cases in which it executes, either on the cloud deployment or local deployment as an executable, without the need of client having to perform some configurations

- The application can change its behaviour if the client provides some specific values during the launch of the application

- The application can recognize or collect data from files residing in external environment

- The local deployment of the application should not expose the source code directly from which the application is written

- The developers on the client side can connect to the locally deployment license server to register the use of the license for themselves

- Database schema can be different if needed for different deployment scenario, and the application should be able to handle this without user effort

- When there is a need to update the database schema, the application is capable of performing database migration by itself without requiring effort from clients

- The application’s metadata and icon can be configured to reflect the details associated with the build version

- Both use cases, which are cloud and local deployment, can be realized using only one codebase

3.2 Project architecture

The project architecture can be explained from the following figure.

Figure 2 shows from a high-level how the application – the license service – can be understood to be operating.

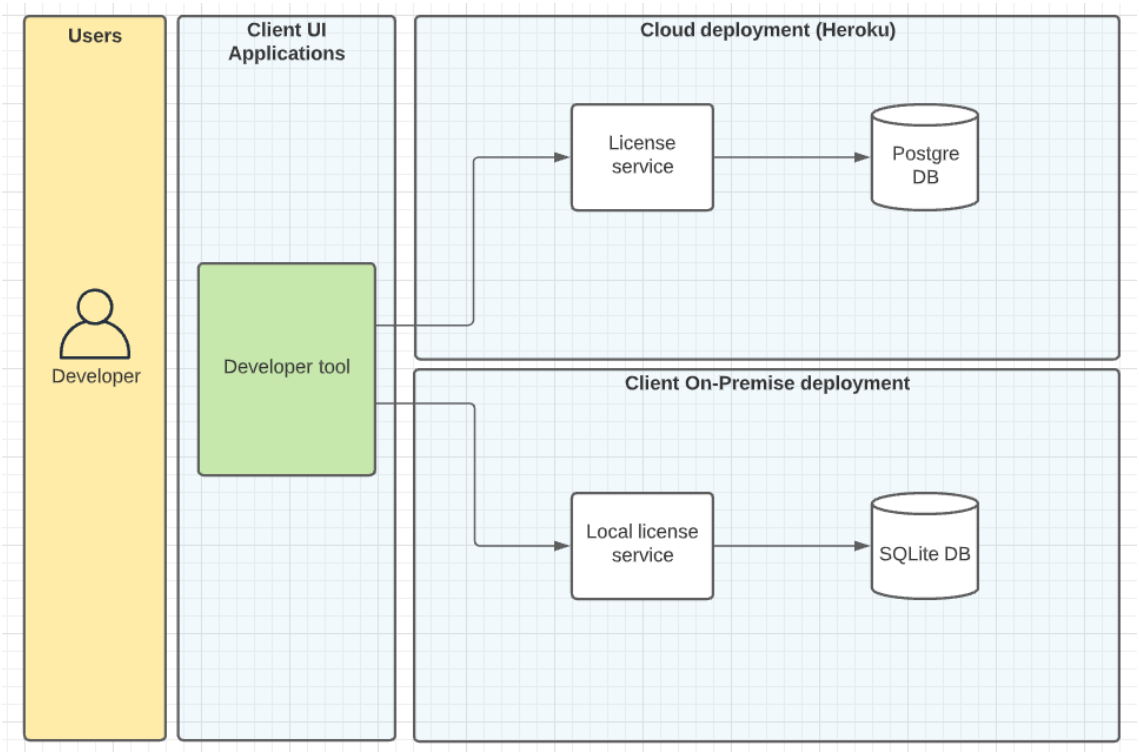


Figure 2. High-level view of license service use cases

As inferred from the figure, there are two different use cases in which the license service can be deployment. The developers are provided with the option of choosing which license service they would like to connect to. For simplicity, the client can always provide outbound internet access that enables developers on their side to connect directly to the cloud hosted license service.

The selection of a service provider is an important part of the cloud-based deployment solution. When the topic of cloud hosting of an application is concerned, there are a multitude of different candidates such as Amazon Web Service, Google Cloud Platform, Microsoft Azure, Heroku, Digital Ocean etc. For

the scope of this thesis research work, Heroku is preferred as it is profoundly easy to host an application on this platform. Almost all needed operations can be configured from the graphical interface dashboard, but such functionalities would be provided also as command line tool. Heroku offers many add-ons which handle a diverse list of different needs, including database management. PostgreSQL can be provisioned from Heroku PostgreSQL add-on, which will serve as the persistence solution for the application in the cloud.

Nevertheless, there are as well cases in which the client is hesitant in allowing this external communication due to for example security issues or firewall setting complexity and so on.

To address that, the second deployment approach which happens in the client's premise can be considered. The two deployment approaches should be mostly identical in terms of underlying core logic, but there is also room for modification or extra configuration which the client can benefit from. Having the client's requirements of effortless deployment in the equation, on-premises approach must select a suitable option for implementing persistence solution, and SQLite is a suitable one. The application carries the responsibility of managing and utilizing the SQLite database, and that the client spends minimal to no effort in maintaining any database server, which is considered optimal in the context of customer experience.

3.3 Project structure

In this section of the chapter, the project structure will be explained in detail. Before the actual start of building the application, a proper and well-defined project structure will assist the software developer in managing different parts of the codebase in a most meaningful manner.

The application's code is organized into different subfolders, in which every single folder holds only the parts that share somewhat a similarity in functionality.

Figure 3 will describe how the structure of the application is organized and defined.

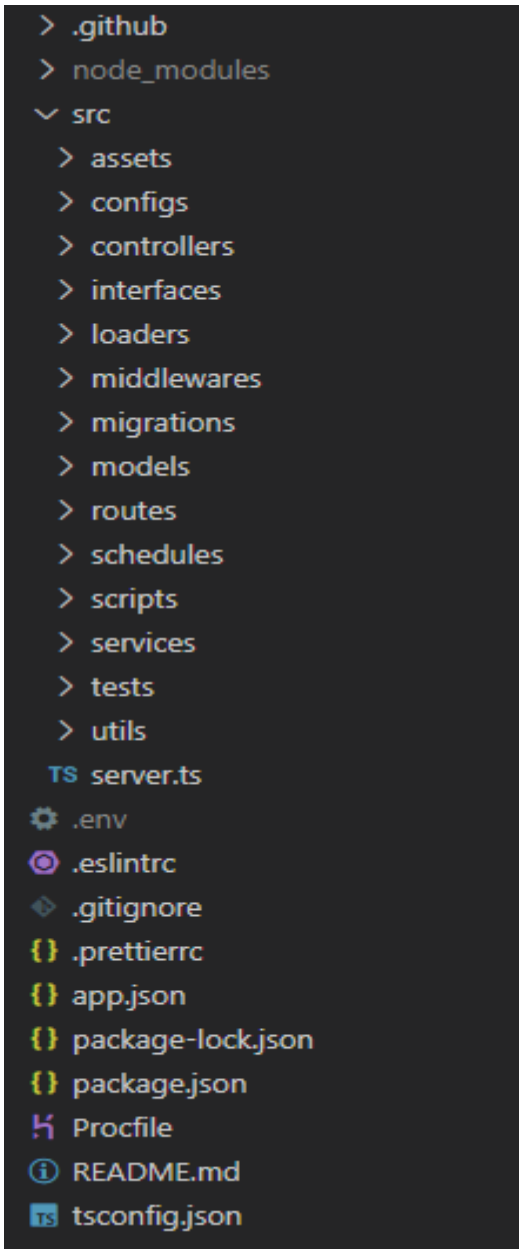


Figure 3. Application/project skeletal structure

The project is developed utilizing Model-View-Controller concept as its foundation. This is further propagated into the skeletal structure of the project, in which the application can be dissected into layers. The main codebase resides in 'src' folder, in which it is divided into smaller segments handling their own responsibility. The application entry point is 'server.ts' file, which handles the instantiation of the application. The sub-division into folders within the main 'src'

folder is to adhere to the S.O.L.I.D principles where each code part has a separate concern which does not overlap with others. Each subfolder is named in such a way that it is self-descriptive of what job it is entitled to. Most important subfolders, which are ‘configs’, ‘controllers’, ‘models’, ‘routes’, ‘services’, ‘scripts’ will be elaborated in upcoming sections.

All configuration detail is stored in ‘configs’ folder, in which it is sub-divided into small and separate parts depending on the running environment. There are four settings in which the application runs, being development, staging, production and local server.

Development configuration applies when the application is run in the developer’s local environment, while staging means that the application is being hosted on the cloud but is considered an unofficial environment only for near-production testing purpose. Production setting will apply when the application is run in the official production environment where end-users are the clients’ developers. Local server configuration will only be available when the application is packaged as an executable and run as is.

Because of the project being a web server, the application would start from the top layer being the routing layer. All routing layer codes are organized into ‘routes’ folder. Their purpose is only to register the API endpoints of the application, the controller which will handle the request and expose them for use when the application runs. The routing layer in some cases can also attach middleware functions to specific endpoints when necessary.

Next, the controller layer, whose code is in ‘controllers’ folder, will take the responsibility to receive client requests. One notable point here is that controller layer has no responsibility to process the request. It passes them to the next layer responsible and afterwards return the responses which are produced by the layer underneath back to the client with appropriate status code. The next layer, which is the service layer, is where the business logic processing takes place. The controller layer in this sense is more like a middleman, passing requests and

waiting for responses to return to client side. Its main job is to re-organize the payload neatly for service layer to process, and to format the response and attach the correct status code depending on the result.

It is without doubt that the service layer would be the most important in the application, where actual business logic is handled. The code for this layer lies in ‘services’ folder in the hierarchy. Due to its important role, the code organized into this folder is highly complex compared to other layers. When the request has been prepared by the controller layer, it will arrive to the service layer and the handling of business logic will start. Depending on the routes, the execution time for the service layer might vary significantly. During the process, it is highly likely the service layer needs to communicate further with the ‘model’ layer, where database transactions are to be executed. After the request’s processing is complete, the service layer will return whatever result available upstream back to the controller layer. Its duty hence will end there, as it does not communicate directly to the client.

The model layer, residing in ‘model’ folder, is often considered the deepest lying layer in the application structure because of its responsibility. Database schema, entity models, database connection and transactional queries are all handled by this layer. When the service layer contacts the model layer, it is either a read or write operation that the service layer is requesting from the model layer to perform on its behalf. The model layer receives necessary data to instantiate the queries, and once they are finished the layer will pass back the result of those queries to the service layer and end its duty. It will not participate in any business logic resolution, but only database-related operations.

The ‘script’ folder does not represent any layer in the application structure. It is, on the other hand, where important script files are stored and awaits to be executed under specific conditions. Some of the scripts handle cron jobs, which are to be run on some particular time, while others are dedicated to support the packaging process of the application into an executable. The script files do not

play a role in the request handling, therefore they cannot be organized or classified as part of any layer in the application structure.
Besides the main codebase, there are files that are associated with other tools whose role is to provide support to the development and deployment of the application afterwards. Some significant files outside ‘src’ folder can be worth mentioning like Procfile, package.json, .eslintrc, .prettierrc, .env.

Procfile is the declarative configuration file that Heroku platform will take into consideration when the application is deployed there. In essence, it instructs Heroku what approach would be most suitable to deploy this application. Package.json stores all dependencies that are used in the project and available metadata related to the project. On top of that, it holds the script commands which will be utilized in later phases for compilation and packaging process.

Eslintrc and prettierrc are configuration files for Eslint and Prettier tool, whose purpose is to aid a developer in making the codebase more organized and neater. When they are in use in conjunction with Visual Studio Code as the IDE, the combination use will enable code formatting automatically to the pre-defined standards from the configuration files, and that the developer needs no extra effort to handle this aspect.

Env file is where all secrets or configuration detail necessary to the application’s runtime in development mode stay. With env file, connection string to databases, secret API keys or similar sensitive information are exempted from the codebase, which improves secret management aspect tremendously. In the next section, the actual implementation or development of the project will be discussed.

3.4 Project implementation

This chapter of the thesis work is dedicated to explaining in detail how the implementation from the code-level point of view has taken place after viewing the project from a high-level perspective in previous chapters.

3.4.1 Express server setup

As the project is a web server application, it is mainly a backend application. The client does not have specific requests for any graphical user interface, but instead a command line tool to perform a specific task. This point will also be covered in a later part.

The project was started with the basic setup for any regular Express application. As it has been mentioned, the file ‘server.ts’ is the entry point of the application, but it is not where the configurations are. Indeed, the code responsible for setting up the Express server stays in ‘loaders’ module.

To be more specific, ‘server.ts’ file will instantiate an empty Express application and pass it to ‘loaders’ module, through which such server instance is provided with all necessary configurations. When ‘loaders’ module has finished attaching settings to the Express boilerplate, it will return the application back to entry point where its listening port will be configured, and the server is considered at that point running. Figure 4 will attempt to visualize this point more closely.

```
// DEPENDENCIES IMPORT
import express from 'express';
import superagent from 'superagent';
import { writeFile } from 'fs/promises';
import path from 'path';
import { Command, Option } from 'commander';

// MODULES IMPORT
import loaders from './loaders/express';
import config from './configs';
import { logger } from './utils/logger';

const port = process.env.PORT || config.default_port;

const startServer = async () => {
  const application = express();

  try {
    const app = await loaders(application);
    app.listen(port);
    logger.info(`License service running at port ${port}`);
  } catch (error) {
    logger.error(`Server instantiation failed! `, error);
  }
};

(async () => {
  await startServer();
})();
```

Figure 4. Application entry point in ‘server.ts’

Inside ‘loaders’ module, the application will process through many different configurations, ranging from importing needed dependencies, mounting API endpoints to setting some universal middleware functions. In figure 5, the ‘loaders’ module will be described in detail.

```
// MODULE IMPORT
import config from '../configs';
import { logger } from '../utils/logger';

// MIDDLEWARE IMPORT
import * as middleware from '../middlewares';

// ROUTES IMPORT
import baseRoutes from '../routes';
import reservationRoutes from '../routes/reservations';

const ExpressLoaders = async (app: Application): Promise<Application> => {
  logger.info('Starting License service...');
  app.use(express.urlencoded({ extended: true }));
  app.use(express.json());

  app.use('/api/v2', baseRoutes);
  app.use('/api/v2/reservations', reservationRoutes);

  app.use(middleware.genericErrorHandler);

  return app;
};

export default ExpressLoaders;
```

Figure 5. Express application configurations in loader module

The application uses express.json() function which replaces the previous well-known alternative which was provided by the body-parser library. Body-parser library, however, has been deprecated. It means that Express now does support natively the functionality to parse the body from client requests without relying on external dependencies.

The most important endpoint which will be of most use for client developers would be the reservation route, through which developers via their tools can send requests to ask for a license. How the route is implemented in routing layer will be explained next in figure 6.

```
// DEPENDENCIES IMPORT
import express from 'express';
import expressJwt from 'express-jwt';

// MODULE IMPORT
import * as reservations from '../../controllers/reservations';
import { getClientJwtKey } from '../../utils';
import config from '../../configs';

// MIDDLEWARE IMPORT
import * as middleware from '../../middlewares';

const router = express.Router();

router.use(middleware.validateHTTPMethod);

/* Only apply JWT checking for cloud setting, otherwise lifted for proxy setting */
if (config.env !== 'proxy' && config.env !== 'test-proxy') {
  router.use(expressJwt({ secret: getClientJwtKey, algorithms: ['HS256'] }));
  router.use(middleware.validateJWT);
}

/* Only apply HMAC checking for proxy setting */
if (config.env === 'proxy' || config.env === 'test-proxy') {
  router.use(middleware.performHmacCheck);
}

router.use(middleware.checkLicenseNumberExistence);
router.use(middleware.validateRequestPayload);

router.post('/', reservations.handleReservationCreation);

export default router;
```

Figure 6. Reservation route in routing layer

The routing layer handles registration of API endpoints and possible middleware function attachments. The idea is clearly visualized in figure 6 where no business logic will take place in this layer. Some validation middleware functions are in use, for example validateHTTPMethod() which allows only specific methods for this endpoint or checking user json web token (JWT) for a valid user identity before proceeding. The endpoint address for the reservation route is /api/v2/reservations, and the request method is ‘POST’. During registration process, the corresponding controller is provided, which is ‘handleReservationCreation’.

On a sidenote, it can be observed that for a particular environment, which is represented through ‘config.env’ value, the application can mount different middleware functions to be used. How this is possible will be covered in later chapter where the packaging process is introduced. Figure 7 will describe the

controller layer code where the client request sent to reservation endpoint will continue.

```
// DEPENDENCIES IMPORT
import { NextFunction, Request, Response } from 'express';
import * as reservations from '../../services/reservations';
import { errorMsg } from '../../utils/errors';

export const handleReservationCreation = async (req: Request, res: Response, next: NextFunction): Promise<void> => {
  let user_email: string | undefined;
  let op: string | undefined;
  let username: string | undefined;

  const ip = req.clientIp!;
  const { license_number, hw_id, src, src_version, host_os } = req.body;
  const payload = { license_number, hw_id, src, src_version, host_os, ip, user_email, op, username };

  try {
    const response = await reservations.createOrExtend(payload);

    let result = {
      status: true,
      message: 'Reservation created/extended successfully',
      reservation: response,
    };
    //@ts-ignore
    if (response.error_type) {
      result = {
        status: false,
        //@ts-ignore
        message: errorMsg(response.error_type),
        reservation: {},
      };
    }
    res.status(200).json(result);
  } catch (error) {
    next(error);
  }
};
```

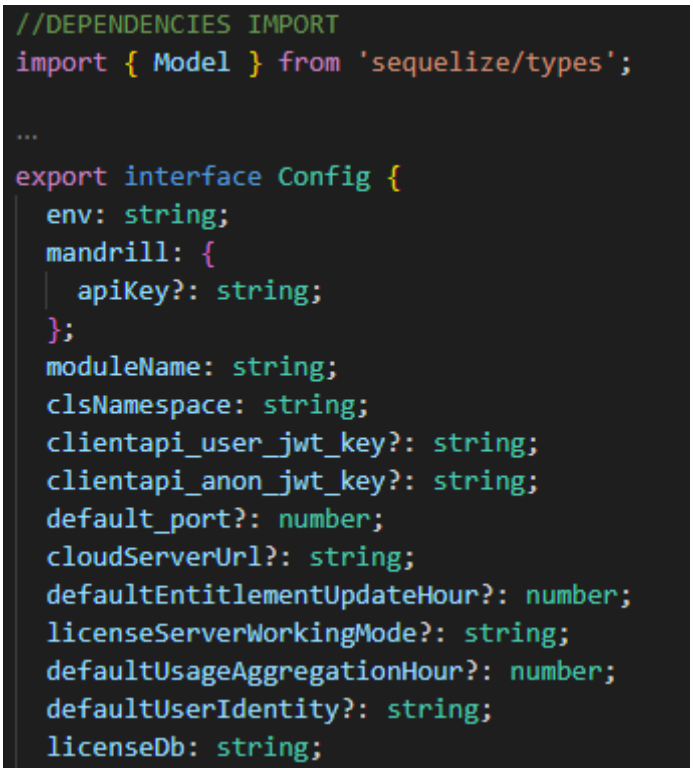
Figure 7. Reservation route in controller layer

From figure 7, the controller is observed as being the man in the middle. It receives the request object passed downstream from routing layer and extracts necessary values out of it. After that, it prepares a new payload package and delivers it to the service layer underneath. It does not handle any business logic, but simply awaits the responses from service layer and then prepare the response payload back to the client with proper status code.

The service layer for reservation endpoint is provided as an asynchronous call in the form of a promise-based operation. Because of that, the controller can take into use the new syntax ‘async-await’, which is merely a syntactic sugar coating for the former ‘then-catch’ syntax that came with the original promise operation. When ‘async-await’ is in use, the developer can take advantage of another great

statement being ‘try-catch’. Regardless of what type of errors that may be returned from the service layer, the controller only needs to detect them in the ‘catch’ block of the statement, and forwards them to the general error handler which is a middleware function that was registered when the Express application was starting. The next function call that is provided by the service layer for the reservation endpoint is ‘createOrExtend’, and this is the function that resolves all business logic associated with the client request sent. In the next section, the service layer will be explored more thoroughly.

It is also worth noting here that since the application was developed with TypeScript, some type bindings were employed to ensure that the data defined or collected stays closely to its intended type. The types that are in use as part of the application originate from two different origins. They can be from the dependencies as built-in typing definitions or they can also be developer-defined types, or custom types. All custom types for the project are stored in ‘interfaces’ module and figure 8 will attempt to provide more details on them.



```
//DEPENDENCIES IMPORT
import { Model } from 'sequelize/types';

...
export interface Config {
  env: string;
  mandrill: {
    apiKey?: string;
  };
  moduleName: string;
  clsNamespace: string;
  clientapi_user_jwt_key?: string;
  clientapi_anon_jwt_key?: string;
  default_port?: number;
  cloudServerUrl?: string;
  defaultEntitlementUpdateHour?: number;
  licenseServerWorkingMode?: string;
  defaultUsageAggregationHour?: number;
  defaultUserIdentity?: string;
  licenseDb: string;
```

Figure 8. Sample custom type definitions for the application

Figure 8 describes how custom types can be declared for specific use in the application. The developer used the concept of interfaces to construct custom type bindings for the data where they are needed. Once declared, the interface object must be exported to be available in other modules, which is in similar fashion as regular JavaScript module export usage. Inside modules where a type needs to be applied to a variable, such interface can be simply imported, and binding can be created.

Figure 9 will continue the explanation on how service layer will handle the business logic from the client request it receives for reservation endpoint.

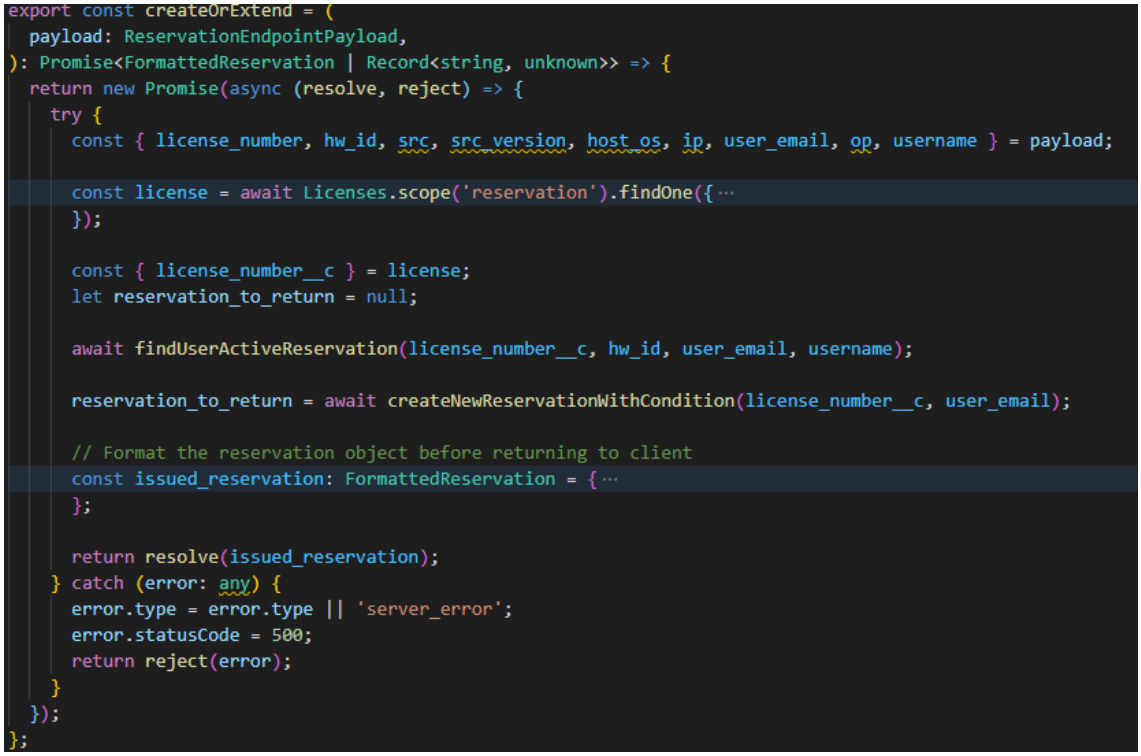


Figure 9. Reservation route in service layer, abridged version

Inside service layer, the main function responsible for the logic, which is createOrExtend, is wrapped as a promise-based operation. This is since service layer is the place where all business logic is resolved, there is a possibility that such logic would require long and heavy computational effort. If the function is to be run synchronously, it might be a blocking operation that makes the server unable to handle incoming requests. By setting the function as a promise, it is

essentially now an asynchronous operation that would be placed into the event queue and subsequently yield back outputs to the event loop.

The payload which has been prepared by the controller layer is destructured into different variables and ready to be used. Next, the license number value is used as a parameter in the invocation of database query searching for the corresponding license. It is worth noticing that the direct communication with the model layer, or database layer, is handled only by the service layer. Routing or controller layer does not participate in this operation as the rule of single responsibility applies. When the database query has returned the needed license data, the logic will proceed searching for the user's currently active license usage, if there exists any. Then, as a next step, a new session or reservation would be created for the user based on his/her email. This would require a write operation to the database to record the user's new license reservation.

Before returning the data upstream to the controller, the information of the license reservation is formatted so that it would not be too verbose and contain irrelevant subsets of data. Formatting the output of business logic resolution is also another duty that service layer needs attention to. If the request is resolved smoothly, service layer will wrap the formatted data object and send it back to controller as a fulfilled promise. In case of unexpected errors, service layer will transport the error object in its response as a rejected promise. The controller will then depend on the outcome of the promise to determine accordingly what message and status code it should send back to the client. Next chapter will provide more insight into the project's database schema and the configuration of the model layer, which is responsible for read/write queries to and from the connected database.

3.4.2 Database schema and configuration

The application is a web server that employs a database as its persistence solution for license usage. Before the database can be properly configured, a schema or how should the model entities be related to each other in a relational

database is needed. In figure 10, the schema for the database in question will be illustrated as an entity diagram.

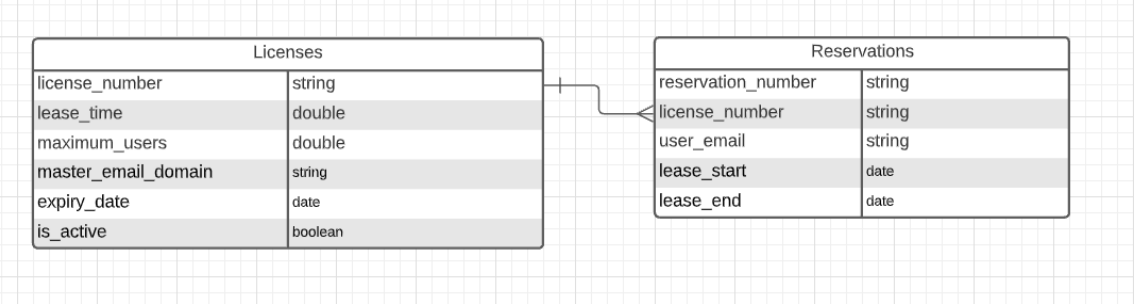


Figure 10. Entity diagram for license and user reservation

From figure 10, the entity diagram shows that the data modelling for the application is simple. It contains only two entities, which are ‘Licenses’ and ‘Reservations’. In ‘License’ model there is license number field which serves as the primary key of the table, and the value is also unique. License number is used to identify license records existing in the database. There are other fields in license model that are important to the whole application logic. Lease time would be used to determine how long a user will get his license reservation for. Maximum users value would mean that a particular license may accommodate at maximum a predefined number of users. Any extra requests that would exceed this limit will result in a refusal from the license server. Master email domain value means that any user whose personal email matches the email domain of the license will be able to request for usage permission. Expiry date and active value are straightforward in meaning in a way that they determine when the license will cease to be valid or whether the license is currently in use or not.

Reservation model can be understood as user-centric data records because these reservations are provisioned for individual use. The table has its primary key or unique identifier with the name ‘reservation number’. In this table, the license number value serves as a foreign key that establishes the relationship between the two entities, which is one-to-many in nature. This would mean one license record may be linked to multiple different reservation records, while one

reservation record is associated with one and only one license. In the entity, user email value emphasizes the individuality of a reservation record in which for any single reservation there will be a user email connected to it. Lease start and lease end values are of type ‘date’, and those represent the chronological values of a reservation. A reservation with a past lease end will not be considered valid anymore and that users will need to request for a new one.

After the database schema has been devised, the application will shape its model layer accordingly. In the project, Sequelize ORM library was used to abstract the construction and usage of the database, in which figure 11 and 12 will explain in more detail on a code level.

```
export let sequelize: Sequelize;

(async () => {
  try {
    if (config.env === 'proxy') {
      sequelize = new Sequelize({
        dialect: 'sqlite',
        storage: `${path.dirname(process.execPath)}/data.sqlite`,
        ssl: false,
        logging: false,
      });
      await sequelize.authenticate();

      const umzug = new Umzug({ ···
      });
      (async () => { ···
      })();
    } else {
      sequelize = new Sequelize(config.licenseDb, {
        ssl: config.dbSSLConfig,
        logging: false,
        dialectOptions: config.sequelizeDialectOptions,
      });
      await sequelize.authenticate();
    }
    logger.info(`Connection to DB authenticated..`);
  } catch (error: any) {
    error.type = 'database_error';
    logger.error(`Could not establish connection to DB`, error);
  }
```

Figure 11. Instantiation of Sequelize instance for database connection

In figure 11, a Sequelize instance is firstly required. After that, depending on the deployment context, the library is instructed to make connection to different SQL

database dialects. If the application server is to be run as a packaged application, Sequelize ORM will create locally a SQLite database file and maintain connection to it. On the other hand, the application when deployed on the cloud will attempt to connect to an existing database via a connection string, and in the scope of the project it is PostgreSQL. Sequelize ascertains that the communication to the database is established properly by invoking an ‘authenticate’ function. It is also noticeable that in the setting for packaged application, the application is using Umzug library to handle database migration for the locally deployed SQLite file. It is, however, not part of the thesis work’s scope and hence will not be investigated further. All the configurations are wrapped in an IIFE function, which would execute automatically when the module is imported by the main application. Afterwards, the Sequelize instance which was provided with needed configurations is exported and is available in other modules.

```
const LicenseSchema: ModelAttributes = {
  license_number: {
    type: DataTypes.STRING(80),
    unique: true,
  },
  lease_time: {
    type: DataTypes.DOUBLE,
    allowNull: false,
  },
  maximum_users: {
    type: DataTypes.DOUBLE,
    allowNull: false,
  },
  master_email_domains: {
    type: DataTypes.STRING,
    allowNull: false,
  },
  expiry_date: {
    type: DataTypes.DATEONLY,
    allowNull: false,
  },
  is_active: {
    type: DataTypes.BOOLEAN,
    allowNull: false,
  },
};

const License = db.define<LicenseInstance>('license', LicenseSchema);

export default License;
```

Figure 12. License model in model layer

In figure 12, the license model entity is transformed from schema to concrete fields and value types supported by Sequelize. This would serve as the backbone for the model layer, so that each entity would reside in its own module and namespace. It would have all necessary fields, values, validation rules etc. configured from the model layer, and once exported it would be ready for handling database queries from service layer.

The service layer would simply import the model it needs to communicate with and invoke corresponding high-level functions attached to the model. Some examples seen previously in service layer would be ‘License.findOne’ function, in which the service layer asks the license model to fetch one record that matches a certain condition. With the usage of Sequelize, it yields several different benefits to the application developer. The developer no longer needs to write expressive raw queries to retrieve the same information regarding license records, and that it would help prevent some vulnerabilities that come with raw SQL queries such as SQL injection attacks or typos during statement preparations. In addition, the developer can attach types to the model entity, which boosts productivity, and that the developer can be certain of the types of the values returned from the model layer as it has been preset.

3.4.3 Command line tool

The license server application does not have a graphical user interface, or frontend representation. Instead, it was configured to provide command line interface tool in case of packaged condition. This section is dedicated to explaining the needed elements to build some CLI commands into the application.

The library used to enable CLI support is Commander.js, which is a popular JavaScript library that help simplifying CLI command construction. Basically, the application would need to be configured in its entry point the commands and necessary options, so that Commander library can be instantiated together with application launch.

```
import { Command } from 'commander';
const program = new Command();

program
  .name('license-server.exe')
  .usage('<command> [options]')
  .command('release', { isDefault: false })
  .description(
    `Early release an active reservation to today's date. Provide either reservation ID or username/user ID`,
  )
  .option('-u, --url <url>', 'URL of running license server. Required.')
  .option('-ri, --reservationId <id>', 'ID of the reservation to be released.')
  .option('-ui, --userId <id>', 'Username/User ID associated with the reservation to be released.')
  .showHelpAfterError(true)
  .showSuggestionAfterError(true)
  .action((options) => {···
  });

program.parse(process.argv);
```

Figure 13. CLI command for license server application

Figure 13 explains how a sample command can be set up and understood by Commander library. Firstly, an invocation of Command class from the library is needed. After that, different values can be provided for each field, for example the command name, its description, what options it would require and the action or handler for what the command should do when it is executed by the user. Then, when it is invoked as a CLI command from the user, the script will attempt to parse the values provided by the user for the corresponding command. If there would be any error, the script will return immediately to the console and notify the user.

In the project's scope, a command was provided so that a user can release a particular reservation early. It comes with command name as 'release', which takes some mandatory options like reservation ID associated with the record to be released prematurely.

The CLI tool is considered the customer-facing frontend of the application server because it would be where users would interact with it. The idea of higher preference of CLI over GUI was originated from the fact that it is more convenient, more reliable, and faster to use. The bundling of simple commands would also require less effort than scaffolding a full-fledged UI with frameworks.

3.5 Project packaging

When the application code has been developed to the extent that a minimum viable product state is achieved, the packaging process of the license application server will proceed. This packaging process has a significant impact on the project's scope, since it will determine how the application can be deployed on the customer's premise.

As previously mentioned, the target deployment environment for the license server is on Windows operating system. The requirement has been that it is a web application deployed and run locally within the customer's private network and should depend on no external installation or libraries to operate. Therefore, the application will be packaged into an executable for Windows with the help of vercel/pkg library that was thoroughly introduced from the theoretical part of the thesis work.

3.5.1 Packaging process

The idea behind the packaging or compiling the application server is to transform a TypeScript codebase to be able to run similarly to a native application developed for Windows. Normally, this is not the case where Node.js applications are often deployed on the cloud in Linux virtual machines. This creates somewhat an interesting solution in a way that the developer does not need to learn C# language or some other programming languages better suited for this particular purpose. He then can utilize his knowledge from JavaScript/TypeScript to create a web application that would work in the same way whereas it is deployed in the cloud or locally. In the next figure, the overall architecture of the packaging will be explained, so that the process from a TypeScript codebase to an end-user executable file on Windows can be understood.

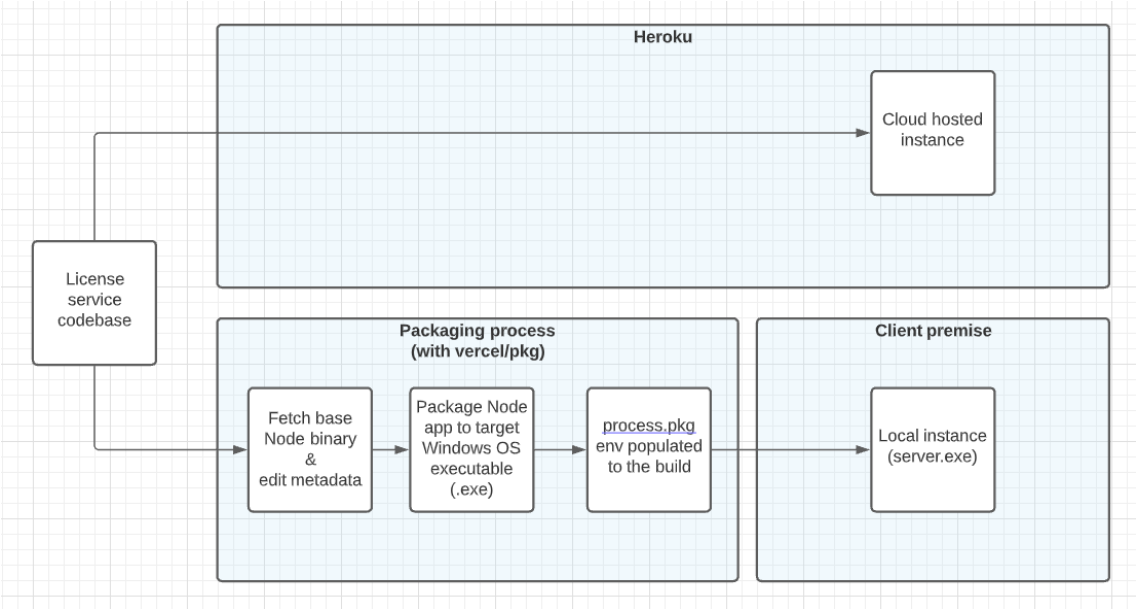


Figure 14. High level architecture of application packaging process

Figure 14 explains how the application can be delivered in different environment. Since cloud deployment is not a major point of concern for the project, it will not be deeply explored in comparison with local deployment approach. Simply, the codebase can be easily deployed on any cloud platform and within the scope of this project Heroku was chosen as the destination which hosts the application. There is no need for extra configuration to make the application work. Heroku as a platform as a service retains the responsibility in managing and keeping the application running as supposed to.

For the same instance in a local setting, the codebase written in TypeScript needs to go through multiple steps so that the final output can be run on a Windows machine. It is worth mentioning that vercel/pkg is a library that compiles JavaScript code and packages it into a target operating system execution file. This essentially means that the tool cannot operate directly on top of a TypeScript codebase. Hence, the application code must first be transpiled from TypeScript to JavaScript and vercel/pkg tool will handle the remaining steps.

When the transpilation step has completed, the packaging tool will attempt to fetch a Node binary image as a base from Node.js official GitHub repository which pkg tool will bundle the source code on top of this image. The process would

repeat every time a packaging process starts. However, developers can instruct pkg to build from source, essentially commanding the library to reutilize an existing Node.js image on the machine for the build step. For this approach to work, there are a few prerequisites needed. Since it will be an existing Node.js image to be reused, pkg tool needs to still fetch it at least one time from the online repository so that it is present locally on the build machine. Then, the downloaded image should be renamed from its original state to signify pkg tool that it can use such binary image instead of fetching a new version from online source on every instance.

By default, the images fetched by pkg tool will be placed in a directory at /.pkg_cache whose exact location differs by operating systems. The name of those binaries is fixed with the format having ‘fetched’ keyword in it, in addition to other information like Node.js version etc. Developers would then need to rename this binary image and modify ‘fetched’ to ‘built’, after which pkg tool will be able to detect the pattern and cease to acquire a new base image on every build. This is used in conjunction with a parameter in the build command which will be explained in a later section.

The metadata and icon of the final output, without any extra configuration, will be default to Node.js environment and its logo. Of course, this is not a desired result. The application would need to have its own metadata details describing what it is and from whom it was built, and on top of that it would need a dedicated logo or icon to represent the owner. This is possible but would not relate directly to pkg tool, as the modification of the executable’s metadata and icon happens to the base Node.js binary image before the source code is bundled into it. To achieve this, the developer used rcedit library to modify the needed information. When the rcedit is provided with the Node.js base image, the developer can use its API to start changing the metadata of the to-be-built executable and its icon.

When the Node.js binary image has been prepared, pkg tool will start traversing the codebase, which is now JavaScript transpiled from TypeScript, and will attempt to include every possible dependency library and reference. It will then

bundle the whole codebase with all necessary modules and apply them onto the base image. This process will last for a duration which varies from codebase to codebase. It is worth iterating that pkg tool however cannot include everything automatically during the build process. For literal requirement calls or static assets, it is necessary that those would be provided to pkg tool in the option inside package.json file so that the tool understands it would need to manually locate those files and bundle them together with the codebase. It is worth mentioning that it is not possible to exclude any module or code section from the packaging process. This behavior would be like that of other containerizing tool like Docker, in which the application is simply put inside a container together with all modules. However, this may be an area of feature enhancement in the future that later version of pkg tool might allow some exclusion of files or modules.

At the end of the build process, pkg tool will inject into the executable some values in the ‘process’ object, namely process.pkg. These values will not exist if the codebase is not packaged by pkg tool. The values of the process.pkg have been described in the theoretical part of the thesis work, and it is worth stating that the existence of these values has far more significance than the actual literal values they carry. By detecting the existence of process.pkg at the initiation of the application, the codebase will know in which setting it is operating without end-users having to specifically provide the context and hence will behave differently if needed.

The outcome of the package process will be an executable for the target operating system, being of extension ‘.exe’ for Windows. The license server application now would be ready to run, without extra configurations. The actual build script and metadata modification example will be explored in the next section.

3.5.2 Packaging script

There are two ways to instruct pkg tool to handle packaging process. On the build machine, pkg can be globally installed and invoked as command line tool. The second approach is done via programmatic API call that pkg tool provides. This will be the approach in which the project is built. In the next figure, the build script which is run by pkg will be elaborated.

```
import rcedit from 'rcedit';
import path from 'path';
import * as pkgFetch from 'pkg-fetch';
const pkg = require('pkg');

// LOAD ASSETS
const iconPath = path.resolve('./src/assets/executable-icon.ico');
const downloadCache = async (pkgTarget: string): Promise<string> => {…
};

(async function build() {
  const pkgTarget = 'node14-win-x64';
  try {
    const cacheExe = await downloadCache(pkgTarget);

    await rcedit(cacheExe, {
      'product-version': '1,0.0.0',
      'file-version': '1,0.0.0',
      icon: iconPath,
      'version-string': {
        CompanyName: 'License Server Project',
        FileDescription: 'Windows Distribution',
        ProductName: 'On-premise License Server',
        LegalCopyright: `© ${new Date().getFullYear()} License Server Project`,
        OriginalFilename: 'license-server.exe',
      },
    });

    const outputExe = path.resolve('./build/license-server');
    await pkg.exec([
      path.resolve('./dist/server.js'),
      ...['--config', './package.json'],
      ...['--target', pkgTarget],
      ...['--output', outputExe],
      ...['--compress', 'GZip'],
      '--build',
    ]);
```

Figure 15. Application packaging script

Figure 15 shows the actual script which is used for the build process. It resides in a JavaScript file and to be executed via a npm script command. Inside, it is essentially an IIFE function which handles everything from patching metadata to calling pkg tool's programmatic API to initiate the build sequence.

The script starts with locating the fetched Node.js binary image on the build machine. Then, the metadata and icon of the end executable can be modified as shown in the figure with rcedit library. Basic information such as product information, company name, file description, icon path etc. can all be changed according to the need.

The final step in the build script is the actual call to the 'exec' API of pkg tool. In here, some options are provided to instruct pkg how it should configure its build process. The value for path.resolve will be the entry point of the actual application in the end executable build. It will be where the executable will search its snapshot filesystem and starts running. The flag '—config' will guide pkg tool to look for any configurations present in the package.json file. It is used mostly for adding static assets and scripts that do not belong to any module dependency.

The option '—target' will be used to determine which target operating system should pkg tool create the end executable for. For this project, 'node14-win-x64' has been selected. This value indicates that the executable is meant for Windows x64, and the Node.js version in use would be 14. The option '—output' specifies the destination directory where the executable is exported, which is for this project the folder 'build'.

The option '—compress' is a feature supported by pkg tool to add compression to the end executable to reduce the size. As a fact, executables built from pkg tool can have a significant size which can range from 10 megabytes up. The size of executables packaged by pkg can be more substantial than for example the same Docker container for the same application. This is since pkg tool includes

the whole Node.js runtime in its executable, and every module or dependency has already been added by then.

The last option that is provided to the API call is '—build'. This is the option used to instruct pkg tool to prefer using an existing Node.js binary image over fetching new image from online repository. Without this option specified, pkg tool will attempt to download from Node.js GitHub repository a new Node.js base image and replace it with the fetched version residing in .pkg_cache on every new build process with the same script. Because the option is provided within the scope of the project, pkg tool will search for the correct version of Node.js image having 'built' keyword in its name, which the developer will need to modify the naming in prior to the build step. The upcoming section will elaborate more on the build commands that are in use in package.json file which will execute the build script.

3.5.3 Build commands and custom options

Once the build script has been prepared, there rises a need for a command to call or execute that script file. In figure 16 some explanations will be provided on how the npm commands are used to assist the packaging process.

Figure 16. Npm commands for build process

From figure 16, it can be observed that there are several steps in a sequence to be run to produce the final executable. Firstly, the 'build' command will handle compiling the codebase from TypeScript to plain JavaScript. Then,

‘fetchBaseNodeBinary’ script will run another script file within the codebase to fetch from Node.js GitHub repository a suitable base image for the Node.js version and the target operating system. Next, the command ‘renameLocalBinary’ will attempt to modify the name of the base image from ‘fetched’ to ‘built’, so that pkg tool can detect and reutilize rather than opting for new download. The combination of ‘fetchBaseNodeBinary’ and ‘renameLocalBinary’ results in another command ‘prepareBinary’, which in essence is only used to run the previous two commands in succession.

The next to last command is ‘buildExecutable’, which is the most important command since this will execute the build script defined previously. All of the explained commands, however, would be components of one single command ‘packageWins’ which will be executed by the build machine. The ‘packageWins’ command will be run in a continuous integration pipeline (CI pipeline) with GitHub Actions hosted runner machine, and the command will start from compiling the codebase, fetching then renaming the Node.js binary image and finally attempting to build the end executable. More information on how the process takes place on GitHub Actions runner will be provided in the next section.

There is also another part within the build process that needs attention, which is the pkg options used to include manually some static assets or components that pkg tool itself cannot do automatically. As seen from the figure, for SQLite library the developer needs to signal a specific node native module so that it can be included into the final build. This is crucial because within the codebase there has been no direct requirement call for the SQLite library, hence pkg would not be aware of such library’s necessity. Furthermore, inside the ‘scripts’ section the database migration scripts are manually added. The reason would be similar to SQLite case in which those scripts are needed but not explicitly required via any modules.

3.5.4 Continuous integration pipeline with GitHub Actions

In order to better automate the packaging process, a CI pipeline is needed. In prior to the existence of any CI process, the procedure to produce the application executable has been manual and error prone. Build steps would take place on the developer's local machine, which may not be suitable and can be littered with wrong configurations. Hence, an automated flow was designed to tackle this issue, and GitHub Actions has been chosen to be the implementation approach for the project.

With GitHub Actions CI pipeline, the build process can transition from manual triggering on the developer's local machine to that of cloud hosted runners when some specific events happen. One major benefit worth mentioning for the consideration of using GitHub Actions pipeline would be that the build process will be less vulnerable to errors and security issues like fetched base images which are corrupted or using a base image on the developer's machine whose machine has been compromised by malware. When run on the cloud, it can be ascertained that the GitHub Actions runner always fetches the right image from the right origin, and it is not cached or reused for subsequent builds as the runner will exit when its job is fulfilled. The next figure will illustrate the actual YAML file to be used by GitHub Actions to run the packaging process for the project.

```yaml
name: Create new license server build release
on:
  push:
    tags:
    - 'v*' # Push events to matching v*, i.e. v1.0, v20.15.10
  workflow_dispatch:
jobs:
  buildLicenseServerForWindows:
    runs-on: windows-latest
    steps:
      - uses: actions/checkout@v2
      - name: Use Node.js 14…

      - run: npm ci
      - run: npm run packageWins

      - name: Upload server build as artifact
        uses: actions/upload-artifact@v2.2.4
        with:
          name: license-server.zip
          path: ./build/license-server.exe

      - name: Create Release
        id: create_release
        uses: actions/create-release@latest
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        with:…

      - name: Download artifact
        uses: actions/download-artifact@v2
        with:
          name: license-server.zip
          path: ./

      - name: Upload Release Asset
        id: upload-release-asset
        uses: actions/upload-release-asset@v1
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        with:…
```

Figure 17. YAML configuration file for GitHub Actions build pipeline.

The YAML file shown in figure 17 resides in a separate folder in the project, which is .github/workflow. In the file, two events are specified that informs GitHub Actions when to run such configuration. One is ‘workflow_dispatch’, which means on manual trigger from GitHub dashboard. The other event is on every new tag that is pushed to the application repository on GitHub. It simply means that whenever new development has taken place and a tag is created to mark the milestone for such development, GitHub Actions will run the YAML configuration to build a new executable from the codebase from the latest pushed tag.

Next, the configuration is set to use the latest Windows version for the runner that handles the build process with command ‘windows-latest’, since it is Windows operating system that is the project’s target environment. The step ‘actions/checkout@v2’ prepares the repository under GitHub workspace which enables access for the workflow. Then, the command ‘npm ci’ is executed to install all dependencies specified in the package-lock.json file with their exact version number.

After this, the most important command in the build workflow is run, which is ‘npm run packageWins’. The command essentially would run in sequence all sub-commands that would firstly fetch the correct Node.js base image, then the image which resides in .pkg_cache directory on the runner machine is renamed so that pkg tool will build from source and not download again from official GitHub repository. As a next step, the base image’s metadata is edited together with the representing icon before it is passed to the pkg tool for patching the application source code onto it. When the bundling process is finished, the executable is located at the directory defined in the build script.

When the packaging process has completed, the next step in the workflow is to upload the executable to GitHub as an artifact. Without uploading this to GitHub, any produced files or data, in this scenario being the executable, will not persist and will be lost after the workflow has finished. The step ‘actions/upload-

artifact@v2.2.4’ is used to perform this task. The executable having the extension .exe will be uploaded to GitHub and compressed to a zip file.

The creation of a release is the next step during the CI workflow after a successful artifact upload has completed. The purpose of creating a release would be to mark the version of the application which the final build represents. This eases debugging issues when specific problems can be traced down to a corresponding build version. It also helps with the documentation of new features or bug fixes in a way that for each release there will be a matching build executable with what it is capable of.

The last two steps in the workflow would be to re-download the compressed version of the executable and upload it to the release as an asset. The action will make the final build available in the release on GitHub and can be accessed from the dashboard. When future need arises for deployment or moving it to some destinations e.g. AWS S3 bucket or some more permanent or proprietary storage, it would be of little effort when new steps can be added into the workflow to enable that.

4 Results and discussions

The end result of the development and packaging of the license application server has been one single executable for Windows operating system that will run wherever without the user needing to concern about preparing the environment in prior. Neither extra installations, libraries nor containerization daemon are required to enable the operation. In the next figure, the appearance and the metadata of the executable will be illustrated.

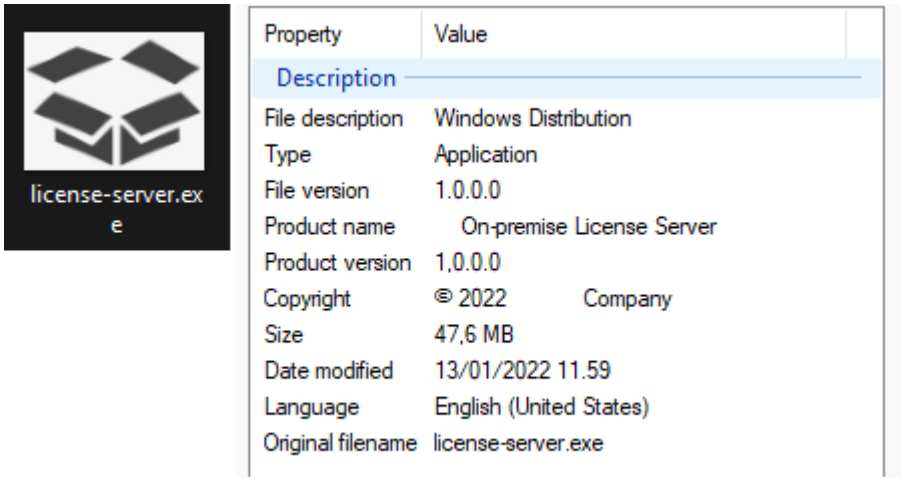


Figure 18. License server executable metadata

From the figure, it can be observed that the application server built and packaged would have a similar appearance to any other ordinary Windows applications that were developed natively. It bears the logo that was provided, and that the metadata was written correctly to show what it represents and from which entity it was built. Such information can be easily configured on every build using the CI pipeline which has been set up for the project.

The executable can be run from the command line or by simply double-clicking to open it like any regular application. When run from the Windows Powershell, some environment variables can be provided to the license server executable to

notify it of wanted behaviour. The executable respects environment variables and will adjust its functionalities accordingly when any valid variable value is provided.

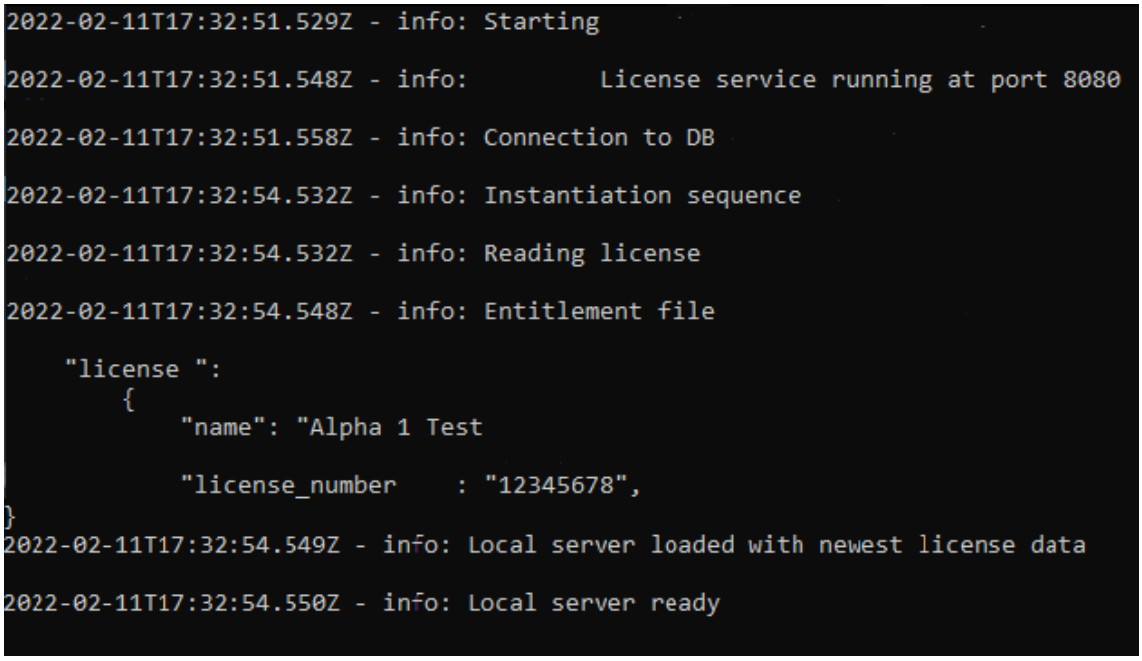


Figure 19. Terminal window of running license server executable

Figure 19 shows when the license server is being run as an executable. It can be observed that all the logs are correctly flushed to the console, and they describe each step during instantiation sequence the license server needs to go through before it is ready to be receiving client requests. Inside the same folder where the executable is, there is a file of JSON extension that contains the sample data of a license to be used, and the license server executable can detect it from its current working directory.

Once it has detected the presence of the license file, it attempts to read the content within and then inserts all found data into the local SQLite file which it created during the start up. The result of the operation will be displayed into the console and that users will know whether the license server has succeeded in loading license data into the database before usage or not. When the instantiation sequence has finished, it will prompt via the logs that it has been ready for connection.

The communication to license server can be instantiated by the client tools, or the user can test the connection by sending requests via Postman tool to the correct endpoint of the license server.

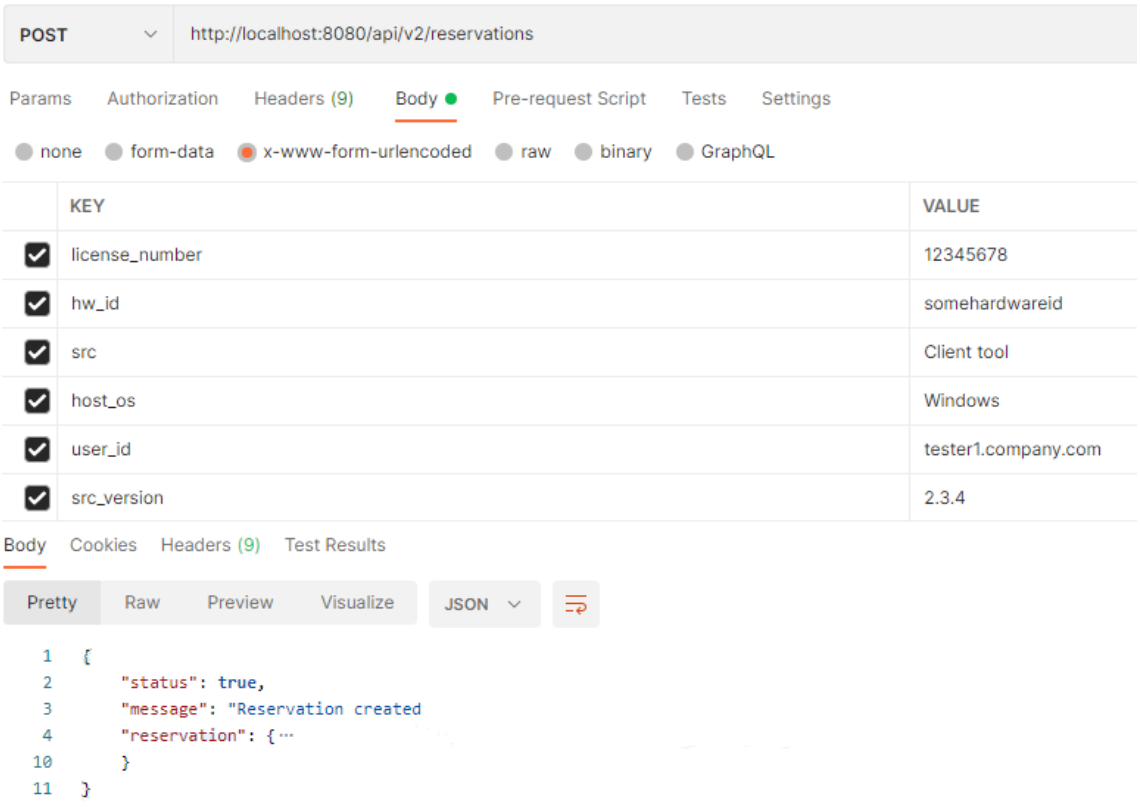


Figure 20. Postman request to license server

Figure 20 shows how a sample POST request can be sent to the license server to determine whether it is running correctly. With the correct values in the request body, which is agreed in advance between the developer who built the license server and the client users utilizing the tools, the communication was successful and that the license server returns the positive result with the reservation data as the response payload. The details of the payload will not be publicized due to privacy restraints, but the idea stays the same.

Once the connection has been tested and verified, the users can start taking client tools into use in conjunction with the on-premises license server packaged

as an executable. The same request that was sent successfully from Postman service can be repeated by the client tools with the accurate payload information for each developer. Hence, the purpose of the license server deployed locally with pkg tool is fulfilled. The exact same version of the codebase can be deployed on Heroku without any packaging process, and it would work in the same manner. The difference lies in the address from which the client tools should be communicating.

The application does not have a graphical interface; however, users can use the pre-built command line tools to interact with it. The next figure shows how command line tool can be explored and taken into use from the user's perspective.

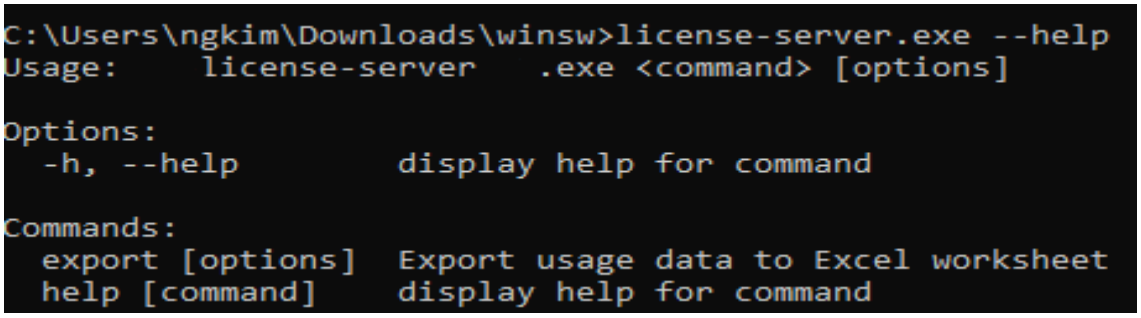


Figure 21. Command line interface of license server

As seen, the commands can be explored by navigating to where the license server executable is and use the option '—help'. All available commands will be displayed, and their own help information can be explored further. This approach provides the interaction that is lacking from the absence of a graphical user interface normally present in other applications. The more features the application is set to include, the more commands are built into it so that any user or administrator can effortlessly interact with the application for various purposes.

5 Conclusions

The objective of the thesis work is to document the journey of the project, and that how the developer went through numerous stages in his development cycle to produce the output that has been agreed mutually with the client users.

By using modern programming languages like TypeScript/Node.js and powerful frameworks and tools like Express and vercel/pkg, the outcome of the project has been achieved in a way that the end users would be able to enjoy the same functionalities that are also available in the cloud deployment via Heroku without having to establish communication to the outside world or public internet.

The thesis work, in this sense, was focused on the technical possibility of delivering a solution that would not be too popular among others yet powerful in terms of functionality. When the question is related to local deployment of a service, the project work has been an example in how it can be achieved by still taking advantage of the JavaScript language meant predominantly for web development for application development on Windows operating system.

6 References

1 David Herron. Node.js Web Development - Fourth Edition [e-book] Packt Publishing; May 2018.

URL: https://learning.oreilly.com/library/view/node-js-web-development/9781788626859/. Accessed 18 Jan 2022.

2 Introduction to Node.js [online] Node.js.

URL: https://nodejs.dev/learn. Accessed 18 Jan 2022.

3 Jethro Magaji. The History of Node.js [online] Section; Aug 2020.

URL: https://www.section.io/engineering-education/history-of-nodejs/. Accessed 18 Jan 2022.

4 Taha Sufiyan. Understanding Node.js Architecture [online] Simplilearn; Dec 2021.

URL: https://www.simplilearn.com/understanding-node-js-architecture-article. Accessed 18 Jan 2022.

5 Developer Survey Results [online] Stackoverflow; 2019.

URL: https://insights.stackoverflow.com/survey/2019#technology. Accessed 18 Jan 2022.

6 Cordenne Brewster. 15 Companies That Use Node.js in 2022 Successfully [online] Trio;

URL: https://trio.dev/blog/companies-use-node-js. Accessed 18 Jan 2022.

7 Josh Goldberg. Learning Typescript [e-book] O'Reilly Media; Sep 2022.

URL: https://learning.oreilly.com/library/view/learning-typescript/9781098110321/ch01.html#idm45887681686192. Accessed 18 Jan 2022.

8 Baeldung. Statically Typed Vs Dynamically Typed Languages [online] Baeldung CS; October 2021.

URL: https://www.baeldung.com/cs/statically-vs-dynamically-typed-languages. Accessed 18 Jan 2022.

9 Jeel Patel. Typescript vs JavaScript: Why Choose Typescript Over JavaScript? [online] Monocubed; Sep 2021.

URL: https://www.monocubed.com/typescript-vs-javascript/. Accessed 18 Jan 2022.

10 Geekflare Editorial. Understanding Difference Between Typescript and JavaScript [online] Geekflare; Sep 2021.

URL: https://geekflare.com/typescript-vs-javascript/. Accessed 18 Jan 2022.

11 Express. Express.js [online].

URL: https://expressjs.com/. Accessed 18 Jan 2022.

12 Besant Technologies. Understanding Difference Between Typescript and JavaScript [online] Besant Technologies;

URL: https://www.besanttechnologies.com/what-is-expressjs. Accessed 18 Jan 2022.

13 Mukul. Middleware in Express.js [online] GeeksforGeeks; Oct 2021.

URL: https://www.geeksforgeeks.org/middleware-in-express-js/. Accessed 18 Jan 2022

14 Jay A. Kreibich. Using SQLite [e-book] O'Reilly Media; Aug 2010.

URL: https://learning.oreilly.com/library/view/using-sqlite/9781449394592/ch01.html. Accessed 18 Jan 2022.

15 What is SQLite [online] SQLite Tutorial;

URL: https://www.sqlitetutorial.net/what-is-sqlite/. Accessed 18 Jan 2022.

16 Chukwuemeka Chima. Introduction to Sequelize [online] Medium; Jan 2019.

URL: https://medium.com/the-javascript-dojo/introduction-to-sequelize-1cbfc2d2d1bf. Accessed 18 Jan 2022.

17 Sequelize [online] StackShare;

URL: https://stackshare.io/sequelize. Accessed 18 Jan 2022.

18 Sequelize [online] SnykAdvisor;

URL: https://snyk.io/advisor/npm-package/sequelize. Accessed 18 Jan 2022.

19 Pkg [online] Npm;

URL: https://www.npmjs.com/package/pkg. Accessed 18 Jan 2022.

20 Developers - vercel/pkg Wiki [online] GitHub Wiki;

URL: https://github-wiki-see.page/m/vercel/pkg/wiki/Developers. Accessed 18 Jan 2022