BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF TELECOMMUNICATIONS
ÚSTAV TELEKOMUNIKACÍ

DEPLOYMENT AND LICENSING OF AN APPLICATION ON GITHUB PACKAGES

NASAZENÍ A LICENCOVÁNÍ APLIKACE NA GITHUB PACKAGES

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR
AUTOR PRÁCE

Anton Misskii

SUPERVISOR
VEDOUCÍ PRÁCE

Ing. David Kohout

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Bachelor's Thesis

Bachelor's study program **Information Security**

Department of Telecommunications

Student:	Anton Misskii	***ID:***	230622
Year of study:	3	***Academic year:***	2023/24

TITLE OF THESIS:

Deployment and Licensing of an Application on GitHub Packages

INSTRUCTION:

The main goal of the thesis is to utilize the version control platform GitHub and the GitHub Packages service for sharing a demo application in the Java language. The application will incorporate the JavaFX graphical interface and Maven build tool. The solution will include an automated build process directly on the GitHub platform, followed by deployment to GitHub Packages for subsequent automatic updates of this demo application.

The demo application will check for the availability of a new version. When the new version is detected, the user will be prompted to update the application. The application will support two deployment types. The first type will be a standalone executable .jar (including necessary packages and libraries). The second type will be a modular application, where all required libraries will be automatically downloaded and are not part of the .jar file.

Another component of the thesis will involve designing and implementing the licensing for this application. Upon the expiration of the provided license, the application's capabilities will be restricted. The license will be cryptographically secured to minimize the possibility of license tampering.

RECOMMENDED LITERATURE:

[1] MASTROPAOLO, Antonio, et al. Toward Automatically Completing GitHub Workflows. arXiv preprint arXiv:2308.16774, 2023.
[2] FERRANTE, Daniel. Software licensing models: What's out there?. IT Professional, 2006, 8.6: 24-29.

Date of project specification:	5.2.2024	***Deadline for submission:***	28.5.2024

Supervisor: Ing. David Kohout

doc. Ing. Jan Hajný, Ph.D.
Chair of study program board

MISSKII, Anton. *Deployment and Licensing of an Application on GitHub Packages*. Bachelor's Thesis. Brno: Brno University of Technology, Faculty of Electrical Engineering and Communication, Department of Telecommunications, 2024. Advised by Ing. David Kohout

Author's Declaration

Author:	Anton Misskii
Author's ID:	230622
Paper type:	Bachelor's Thesis
Academic year:	2023/24
Topic:	Deployment and Licensing of an Application on GitHub Packages

I declare that I have written this paper independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the paper and listed in the comprehensive bibliography at the end of the paper.

As the author, I furthermore declare that, with respect to the creation of this paper, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll. of the Czech Republic, Section 2, Head VI, Part 4.

Brno ....

author's signature*

*The author signs only in the printed version.

ACKNOWLEDGEMENT

I would like to thank my bachelor's thesis supervisor, Mr. Ing. David Kohout, for his expert guidance, consultations, patience, and valuable suggestions for the thesis.

Contents

List of Figures

Listings

ABSTRACT

The work focuses on the versioning platform GitHub, specifically its GitHub Packages service. The goal is to automate the build process of a demo JavaFX application using the JavaFX graphical interface and the Maven tool on the GitHub platform. This includes deployment on GitHub Packages and ensuring the application can be updated efficiently. Additionally, the application must incorporate a system for versioning and licensing to protect the software. Another key objective is to describe and implement the modular Java system introduced in Java 9, which offers enhanced possibilities for application development and distribution. By implementing these systems and deploying the application on GitHub Packages, the project aims to create a comprehensive methodology for developing modern applications, integrating security systems, maintaining a reliable application versioning system, and continuously delivering and distributing applications through popular services.

KEYWORDS

Application delivering and distribution practices, Application versioning resolution, GitHub Actions, GitHub Packages, GitHub Services, Java Platform Module System, Licensing models

ABSTRAKT

Práce se zaměřuje na platformu pro verzování GitHub, konkrétně na její službu GitHub Packages. Cílem je automatizovat proces sestavení ukázkové Java aplikace pomocí grafického rozhraní JavaFX a nástroje Maven na platformě GitHub. To zahrnuje nasazení na GitHub Packages a zajištění, aby bylo možné aplikaci efektivně aktualizovat. Kromě toho musí aplikace obsahovat systém pro verzování a licencování k ochraně softwaru. Dalším klíčovým cílem je popsání a implementace modulárního systému, který byl zaveden v Java 9, který nabízí rozšířené možnosti pro vývoj a distribuci aplikací. Implementací těchto systémů a nasazením aplikace na GitHub Packages projekt sleduje vytvoření komplexní metodologie pro vývoj moderních aplikací, integraci bezpečnostních systémů, udržování spolehlivého systému verzování aplikací a kontinuální doručování a distribuci aplikací prostřednictvím populárních služeb.

KLÍČOVÁ SLOVA

Praxe dodávání a distribuce aplikací, Řešení verzování aplikací, GitHub Actions, GitHub Packages, GitHub služby, Java Platform Module System, Licenční modely

Typeset by the thesis package, version 4.09; https://latex.fekt.vut.cz/

ROZŠÍŘENÝ ABSTRAKT

Tato bakalářská práce si klade za cíl podrobně prostudovat moderní postupy vývoje JavaFX aplikací, s důrazem na distribuci, správu verzí, licencování a využití Java modulů. Jako efektivní řešení distribuce aplikací je prezentováno využití platformy GitHub a jejích služeb. Díky nové službě GitHub Packages je možné publikovat artefakty aplikací přímo na GitHub spolu se zdrojovým kódem, což přináší výhodu kombinace kódu aplikace a jejích artefaktů na jednom místě, což je pro uživatele i vývojáře velmi užitečné. Klíčovou roli v automatizaci procesu aktualizace zdrojového kódu aplikace a zároveň zveřejňování nových verzí balíčků, sehrává služba GitHub Actions, která umožňuje definovat sekvenci kroků pro konkrétní události na GitHubu.

Dále se práce zaměřuje na detaily licenčních modelů a implementuje spolehlivá kryptografická opatření pro kontrolu šíření demo aplikace a ochranu určitých funkcí. Podrobně je rozebrán proces vývoje, výhody a potenciál distribuce modulárních Java aplikací, které byly představeny v Javě 9. Součástí výzkumu je také analýza bezpečnostních aspektů a praktická implementace licenčního serveru založeného na Spring Rest, který zajišťuje generování a ověřování licenčních klíčů. Tento licenční server je navržen tak, aby byl flexibilní a bezpečný, což umožňuje jeho použití i pro jiné aplikace mimo hlavní projekt.

Tímto praktickým přístupem a konkrétními strategiemi si tato práce klade za cíl vybavit vývojáře nezbytnými nástroji pro úspěšné navigování a překonávání problémů, které se vyskytují v moderních vývojových prostředích softwaru. Důraz je kladen na integraci automatizace a bezpečnosti do vývojového procesu, což přispívá k efektivitě a spolehlivosti výsledných aplikací.

První kapitola této bakalářské práce se podrobně věnuje popisu nejdůležitějších nástrojů a služeb pro vývoj Java aplikací, jako je Maven, který slouží ke správě Java projektů. Maven je odpovědný za strukturu Java projektů, definování závislostí projektu a jejich automatické stažení, sestavení aplikace do spustitelných Java archivů a v rámci této práce se používá také k zveřejnění sestavených artefaktů do GitHub Packages. V této kapitole jsou také důkladně popsány systémy kontroly verzí, různé přístupy k nim a prozkoumány výhody a nevýhody každého systému v konkrétních situacích. Velký důraz je kladen na distribuční systém kontroly verzí Git a práci s ním. Další platformou, která je zkoumána v této kapitole, je GitHub a jeho služby GitHub Actions spolu s GitHub Packages, u kterých jsou popsány principy práce s těmito službami, jejich komponenty a způsoby využití jak samostatně, tak i společně. Na konci této kapitoly je příklad nastavení GitHub Packages spolu s Apache Maven a ukázka výsledného GitHub Packages registru.

Druhá kapitola práce se věnuje popisu modelů softwarových licencí, jejich porovnání a kritériím výběru a návrhu vhodných modelů softwarových licencí. Konkrétně se

zaměřuje na subscription model a trial software model pro demo JavaFX aplikaci na základě její funkcionality a požadavků na kontrolu distribuce. V této kapitole je také představen diagram implementace softwarové licence do demo JavaFX aplikace s podrobným popisem každého kroku. Dále je v této kapitole navržena Spring Restful aplikace pro správu licenčních klíčů, která umí komunikovat se základní aplikací pomocí definovaného API. Tato aplikace řeší vytvoření licenčních klíčů jak pro delší období, tak i pro zkušební dobu na základě emailu uživatelů. Z emailu vytvoří hash pomocí algoritmu SHA-256, který pak podepisuje privátním klíčem pomocí ECDSA algoritmu a následně dekóduje vytvořený podpis pomocí base64, čímž vytváří licenční klíč. Aplikace také zajišťuje validaci licenčních klíčů. Přijímá POST request, který obsahuje email uživatele a licenční klíč, vytvoří z emailu hash pomocí SHA-256, dekóduje licenční klíč a ověří tento klíč pomocí ECDSA a veřejného klíče. Rovněž umí pracovat s vypršením a obnovením doby platnosti licenčních klíčů. Podstatné algoritmy generace a validace klíčů jsou popsány pomocí blokových diagramů.

Třetí kapitola bakalářské práce popisuje obecnou strukturu JavaFX aplikací a také konkrétní demo aplikaci. Demo aplikace představuje todo list, který je propojený s databází pro trvalé uložení profilů uživatelů a jejich úkolů. Tento systém umožňuje efektivní správu úkolů, přičemž každý uživatel má svůj vlastní profil s individuálním seznamem úkolů. Aplikace rovněž obsahuje mechanismus pro kontrolu dostupnosti novější verze prostřednictvím GitHub API, čímž zajišťuje, že uživatelé vždy používají nejaktuálnější verzi softwaru. Komunikace s GitHub API umožňuje aplikaci zjistit poslední verzi GitHub balíčku určeného pro aplikaci a případně uživatele informovat o možnosti aktualizace.

Poslední podkapitola se zabývá integrací softwarové licence do aplikace. Uživatelé mohou získat trvalý licenční klíč na určitou dobu, který zadávají ve speciálním okně aplikace. Alternativně si uživatelé mohou vygenerovat zkušební licenci, pokud dosud žádný licenční klíč nemají. Zadaný licenční klíč je uložen do lokální databáze v neověřeném stavu a aplikace jej dále používá. Zkušební licence umožňuje uživatelům vyzkoušet plnou funkčnost aplikace a rozhodnout se, zda licenci prodlouží na delší období.

Pokud má uživatel při přihlašování do aplikace již zadaný licenční nebo zkušební klíč, aplikace odešle HTTP POST požadavek s licenčním klíčem a e-mailem uživatele licenčnímu manažeru, který ověří platnost klíče. Na základě výsledku ověření se stav licenčního klíče aktualizuje v lokální databázi, což uživateli umožní přidávat k jednotlivým úkolům prioritu. Tato priorita se projeví na barvě každého úkolu, čímž se zlepší přehlednost a usnadní řízení úkolů podle jejich důležitosti. Pokud uživatel nemá žádný klíč nebo má neplatný klíč, tato funkcionalita mu nebude dostupná.

Aplikace také umí sama zkontrolovat platnost licenčního klíče v případě nedostupnosti licenčního serveru. Pokud byl klíč již jednou ověřen, aplikace využije poslední stav uložený v databázi a zkontroluje, zda platnost klíče nevypršela. Pokud je klíč stále platný a nebyl zrušen, aplikace umožní přístup ke všem funkcím. Tato funkcionalita zajišťuje, že uživatelé mohou nadále používat aplikaci i v případě dočasné nedostupnosti licenčního serveru, což zvyšuje spolehlivost a dostupnost aplikace pro koncové uživatele.

Poslední kapitola se věnuje vývoji modulárních Java aplikací a jejich distribuci. Kapitola popisuje základní vlastnosti modulů, jejich výhody oproti standardním Java aplikacím a podrobně zkoumá vzájemnou spolupráci modulů. Základem pro tuto kapitolu je Java aplikace popsaná ve třetí kapitole, která je během procesu migrace rozdělena do několika samostatných modulů, z nichž každý odpovídá za určitou funkcionalitu aplikace. Tyto moduly jsou propojeny na základě osvědčených postupů při vytváření modulárních aplikací.

Jednou z hlavních výhod modulárních Java aplikací je možnost vytváření vlastního runtime prostředí, které obsahuje pouze nezbytné moduly pro provoz aplikace a může být distribuováno uživatelům. S vlastním runtime prostředím uživatelé nepotřebují mít předinstalovanou Javu, protože všechno potřebné pro spuštění aplikace a její funkčnost je již součástí tohoto prostředí. Tím se výrazně zjednodušuje proces instalace a nasazení aplikace, čímž se zvyšuje uživatelská přívětivost a snižuje riziko problémů spojených s kompatibilitou různých verzí Java Runtime Environment (JRE).

Kapitola také zdůrazňuje další výhody modulárního přístupu, jako je zlepšení bezpečnosti a údržby kódu. Díky modularitě lze jednotlivé části aplikace aktualizovat a testovat izolovaně, což zjednodušuje identifikaci a opravu chyb. Moduly mohou mít jasně definovaná rozhraní a závislosti, což usnadňuje jejich správu a minimalizuje riziko nežádoucích interakcí mezi různými částmi aplikace.

Výsledky práce zahrnují vytvoření demonstrační JavaFX aplikace a její modulární alternativy, jejichž automatická distribuce je realizována pomocí platformy GitHub a jejích služeb GitHub Packages, GitHub Actions a Releases.

Při aktualizaci původní aplikace se spouští workflow, které vytvoří spustitelný Java archiv (JAR) a následně ho publikuje na GitHub Actions s novou verzí, čímž umožňuje uživatelům stahování aplikace přímo z GitHubu. Při aktualizaci modulární aplikace dochází k automatickému vytvoření Java runtime image, který obsahuje pouze potřebné knihovny, a je distribuován do složky Releases na GitHubu společně s vytvořením Java balíčku pro každý modul aplikace. Tento proces zajišťuje, že uživatelé mají vždy přístup k nejnovější verzi aplikace s minimálními nároky na ruční zásahy, což přispívá k efektivní a bezproblémové distribuci.

Pro potřeby správy licencí, byla také vytvořena Spring Rest aplikace, která slouží jako licenční server pro hlavní aplikaci a provádí veškeré ověřování a vytváření licenčních klíčů. Tato aplikace byla navržena s důrazem na bezpečnost a modularitu. Vytvoření této samostatné aplikace přispělo ke zvýšení bezpečnosti aplikace, správnému rozdělení odpovědností a umožňuje použití daného licenčního serveru i pro jiné aplikace. Licenční server je zodpovědný za generování a ověřování licenčních klíčů, což zajišťuje, že přístup k aplikaci mají pouze autorizovaní uživatelé.

Prostřednictvím výzkumu provedeného v této práci bylo dosaženo komplexního porozumění moderním praktikám vývoje aplikací, které zahrnují oblasti jako automatické doručování a distribuce, správa verzí, správa licencí a modularizace. Získané poznatky poskytují nezbytné nástroje k překonávání výzev spojených se současným vývojem softwaru a k využití plného potenciálu nově vznikajících technologií.

Introduction

When developing modern applications, certain aspects such as subsequent distribution to end users, maintaining a reliable application versioning system, and controlling application usage through licensing models play significant roles. The goal of this thesis is to demonstrate an approach to solving these problems in the process of developing a JavaFX demo application together with Apache Maven, using modern technologies and best practices.

Distributing applications simultaneously with application updates is a crucial aspect of modern development. Utilizing the Git version control system and the GitHub platform offers a comprehensive solution. Noteworthy in this context is the GitHub Packages service, provided by GitHub, which is responsible for hosting and managing packages, including containers and other dependencies [1]. This service was first introduced as part of the GitHub versioning platform in 2019. In this bachelor's thesis, the configuration for publishing application artifacts to GitHub Packages is demonstrated, along with an overview of the key features and usage of GitHub Actions to automate the processes of building and deploying application artifacts.

Moreover, the study provides a review of the main existing licensing models, introduces the key criteria for their selection, and describes an example of implementing licensing using a third-party server that controls licenses. The proposed licensing model contains cryptographic protection that minimizes the possibility of license forgery.

The next important concept discussed in this work is Java modules, a concept introduced in Java 9 that revolutionized the development and distribution process of Java applications. It emphasizes the advantages of modular applications, the migration process from standard Java applications to modular ones, and new possibilities for distributing modular applications.

Through the exploration undertaken in this work, a comprehensive understanding of modern application development practices is achieved, covering areas such as automatic delivery and distribution, version control, licensing management, and modularization. The insights gained provide the necessary tools to navigate the challenges inherent in contemporary software development and to leverage the full potential of emerging technologies.

1 Tools and services

This work primarily involves the demonstration and practical implementation of various development services and tools to enhance the development process. This section describes the fundamental principles of working with these services and tools, as well as to provide a practical demonstration. It includes a description of the Maven tool and its key features. Following that, there will be a description of version control systems, particularly Git. The last two subsections will be devoted to GitHub and its services. Specifically, GitHub Packages, where an example of its usage will be provided, which is one of the main goals of this work.

1.1 Maven

Maven is a build automation tool primarily used for Java projects. The Maven project is hosted by the Apache Software Foundation and was formerly part of the Jakarta project. Maven has the standard way to build the projects, a clear definition of what the project consisted of, an easy way to publish project information, and a mechanism for distributing JAR files among multiple projects [2]. Maven uses a Project Object Model, POM for short. It is an XML file that contains information about the project and configuration details used by Maven to build the project. Components that can be specified in the POM are the project dependencies, the plugins or goals that can be executed [3].

1.1.1 Features

Maven offers developers comprehensive support, starting from project creation to the compilation of the project into executable files. The basic features and options it includes to provide this support are [4]:

- **Simple project setup**
  It means that there is no need to create the project structure. When creating a new project using Maven, one can simply select the appropriate Maven archetype, a predefined project structure, and Maven will automatically generate the basic project that meets the requirements.
- **Project structure consistency**
  This feature is closely related to the previous one: using Maven for creating projects helps developers across the world because Maven standardizes and enforces project structure, making it easier to understand and work with different projects created with Maven.
- **Dependency management**

Most modern applications require the integration of external code, often referred to as external libraries. These libraries enable developers to enhance the functionality of their applications without having to develop their own solutions. Typically, multiple such libraries are employed in a single application, significantly speeding up the development process by reducing the time spent on resolving specific tasks.
Maven simplifies the process of exporting libraries/dependencies, making it both convenient and fast. Developers only need to search for the dependencies in Maven repositories. They can then select the appropriate version and import the dependency into the automatically generated pom.xml file. Maven downloads these dependencies and stores them in the user's local repository. With Maven, developers can easily update the versions of all their dependencies. When a new version is released in the Maven repository, the pom.xml file will contain a notification indicating the availability of a new dependency version.

- **Build lifecycles**
  Build lifecycles in Maven allow developers to build and distribute a particular project in a predefined way. Maven has three built-in build lifecycles: default, clean, and site . The default lifecycle manages project deployment, clean manages project cleaning, and site creates the project's website with project documentation, and adds standard reports about the state of the project's development. Each build lifecycle has own set of goals which must be fulfilled at the end of it. Examles of common Maven goals include: compile, test, package, and install.
  Each lifecycle in Maven is made up of phases that define specific actions and has own set of goals. For example, the phase "validate" is used to validate that the project is correct and that all necessary information is available. Lifecycle phases can be executed sequentially or in any order [5].
- **Plugins**
  Plugins define implementations for various goals and can be used to modify existing goals. Maven provides two types of plugins: build plugins, which are executed during the build process and should be configured in the <*build*> element of pom.xml, and reporting plugins, which are executed during the site generation process and should be configured in the <*reporting*> element of pom.xml [6].

1.2 Git

To understand what Git is, how it works, and why developers may need to use it, one must first comprehend the problem of a Version Control System (VCS), as this is crucial thing that Git is based on. A Version Control System is a system that helps users manage the development state by saving changes of they projects in a specific file or set of files over time [7]. It is convenient for many people to access the oldest versions to their projects, not only to track all differences from the current version but also to control the entire development process and debug new features or changes in the project.

1.2.1 Version control systems

Version control systems can be devided into multiple approaches [7]:

- **Local Version Control System** (LVCS). In this system, there is a simple local database that stores all changes to files under revision control. One of the most popular LVCS systems is The Revision Control System (RCS), which is still distributed with many computers today. RCS can manage multiple revisions of files, automates the storing, retrieval, logging, identification, and merging of revisions, and it also is useful for text that is revised frequently, including source code, programs, documentation, graphics, papers, and form letters [8]. However, there is one major limitation: in a Local Version Control System, people cannot easily share their files with other developers to collaborate on a project. Therefore, VCSs can be useful for local projects but are not intended for projects that involve more than one person.
- **Centralized Version Control System** (CVCS) were developed to deal with problem of sharing data within developers, that can't be done with LVCSs. CVCSs systems have a server that stores all versioned files. These systems have a hierarchical structure managing access rights for each collaborator, ensuring everyone is aware of their responsibilities. A typical Centralized Version Control Workflow is [13]:
  - Pull down any changes other people have made from the central server.
  - Make your changes, and make sure they work properly.
  - Commit your changes to the central server, so other programmers can see them.

  The main disadvantage of this approach is the absence of a local copy of repository, consequently resulting in the inability to work offline.
- **Distributed Version Control System** (DVCS), such as Git, are a type of version control system where users can create full local copies of repositories, including their entire history, from remote servers. Also DVCSs enables

developers to work with multiple repositories simultaneously. This approach resolves numerous issues associated with remote servers and significantly improves the speed of repository-related actions. Here are some of the most crucial advantages of DVCSs over Centralized Version Control [13]:

- Committing new changes can be done locally without pushing them to the remote server. This allows you to thoroughly test all the changes before pushing them to the remote server.
- Because each programmer has a complete copy of the project repository, they can share changes with just one or two other people to gather feedback before presenting the changes to everyone.
- You can work with your repository offline, and then, when you have internet access, you can push your work to the remote server easily.

1.2.2 Storing data

Git has the different approach in storing data then others VCS. Where other VCS would save information about files changes as a set of all current files and the changes made to each file over the time. Git creates snapshots of a file systems, so when user commit new changes, Git saves files state in this moment and stores the reference to new snapshot. If some is not changed, Git doesn't store these files again, it just makes the reference to the previous file that have been already stored.

1.2.3 Local operations

For faster development Git and DVCS uses local storage to have access to all files and metadata without connection to remote server. One of the main advantages of this approach is the possibility of working offline. Developers just need to have a local repository copy to work with it. This feature is especially valuable when the remote server is not available. Developers can continue working without waiting for server accessibility, and when the server is accessible again, they can upload all their changes.

1.2.4 File's states

In Git, there are three possible file states: "**modified**", "**staged**", and "**committed**". When a developer initializes Git in a local directory with the *git init* command and makes changes to files, the status of those files turns to "**modified**". This indicates differences between the current version of the file and the original version saved in the last snapshot.

To apply changes from this current version, the developer needs to stage the file with the *git add* command, which moves the files to the staging area – an intermediate space where commits can be formatted and reviewed before finalizing the commit [10]. In this staged state, the file is prepared to be committed by the *git commit* command to the next snapshot, or it can also be reverted if necessary.

The final state of files in Git is "**committed**". When files are committed, it means they are safely stored in the local database, and the developer will always have access to them.

1.2.5 Branching system

Git utilizes the branching system, which allows developers to create new branches from the main branch of the repository in their local copy and then upload their local branch changes to a remote repository. The main advantage of using the branching system is the structured development process they provide. Using a new branch for each change in the code ensures that developers always have a clear overview of the development of new features, as each feature is divided into its own dedicated branch. As all branches are fully independent of each other, developers can use them to testing new features without affecting the main branch. If the new feature works, developers can easily merge it into the main branch. If, for some reason, a developer doesn't want to implement the new feature into the main branch, then the branch can easily be deleted.

1.2.6 Git workflows

Because of Git's distributed nature and superb branching system, an almost endless number of workflows can be implemented with relative ease. There are two examples of git workflows:

The Centralized workflow is the simplest model where developers push their local changes directly to the main branch of the remote server. One key thing developers need to remember is that Git does not allow you to push new changes if someone else has pushed changes since the last time you fetched [11]. This precaution helps prevent conflicts between local data and remote data, ensuring that data pushed to the server does not conflict with changes made by others.

The Integration Manager workflow involves a designated integration manager who is responsible for committing changes to the main repository [11]. In this workflow, developers fork the main repository on platforms like GitHub or GitLab. They then clone the main repository to their local system, create branches for each feature, commit and push those changes to the fetched repository, and submit a merge request. The integration manager reviews the code and determines whether the changes can be integrated into the main repository. If everything is good, branch with new features will be integrated into the main branch of repository. In this workflow developers and integration manager need to control conflicts. To successfully merge code into the main branch, developers need to incorporate any other changes that have occurred since the last fetch to their local main branch and perform a feature branch rebase to the latest version of local main branch.

1.3 GitHub

GitHub is a code hosting platform for version control and collaboration, founded on Git, and delivered through a software as a service (SaaS) business model [12]. It allows people to work together on projects from anywhere using Git functionality.

1.3.1 GitHub use cases

GitHub is used to store, track and collaborate on software projects in a number of different contexts [12]:

- **Businesses**
  Many big companies use GitHub as a platform to host their products. It offers numerous advantages because different developers within the company can work on the product simultaneously and maintain control over the development process. GitHub also provides a variety of useful features that can be used during code review and safety merging changes.
- **Open source software development**
  That case doesn't really differ from the previous one, but in this case, people share their projects with the entire Internet, and anyone who wants to collaborate can do it. Sometimes big companies do the same, giving some tasks to open source. Git also has a system that encourages developers to contribute to other projects.
- **Non-programmer**
  Some people can use GitHub as a version control system for their documents or multimedia because Git ensures that committed files never disappear, and GitHub is a right and convenient place to access such files.

1.3.2 GitHub as a collaborative coding platform

GitHub provides extensive opportunities for collaboration among developers in creating and making changes to code. The key tool in this process is pull requests, representing the initial stage of integrating new code into the production [14].

The entire process of introducing new code on GitHub is flexible and easily customizable. The platform offers options for configuring notifications for updates in the repository and an extensive code review process. This process involves the review of written code by other team members, who can leave comments on specific code fragments, initiate new discussions, and make decisions regarding the approval of changes. After the successful completion of the code review process, the code can be integrated. A Developer can request code reviews from specific colleagues, selecting them as code reviewers [15].

While the pull-based development workflow introduces valuable opportunities for community engagement, it is not without its drawbacks. The process of reviewing pull requests by developers can be time-consuming and does not guarantee a hundred percent quality of the final code. To address this, GitHub Actions have been developed and implemented. Configuring these actions can streamline the work for repository administrators, as they are capable of automating the routine testing process [16].

1.3.3 CI/CD pipelines

To better understand GitHub Actions and why developers use it, it's helpful to know about the CI/CD method and its advantages. CI/CD stands for Continuous Integration and Continuous Deployment, these terms highlights the core aspect of this method. Continuous Integration is a process of integrating code changes between team members that are automatically build, tested and merged with the existing code. When Continuous Delivery is the process of safely and quickly delivering a product to the user, allowing for continuous feedback and the ability to obtain information related to ongoing changes.

Without integrating a CI/CD pipeline into a project, all testing, building, and merging steps need to be performed manually. This results in an increased workload for repository maintainers, who must handle communication, code reviews, contributor license agreement issues, testing new changes, and the explanation of project guidelines [18].

With CI/CD pipelines, developers can establish a mechanism that automates all these steps. As a result, CI/CD pipelines assist developers in enhancing the quality and productivity of making changes to their products [16].

1.3.4 GitHub Actions overview

GitHub provides its own solution for implementing CI/CD workflows directly on the platform called GitHub Actions. Developers can program their own workflows and share them within the GitHub community via the GitHub Marketplace [16]. Workflows are defined in the .github/workflows directory and use YAML syntax. In each repository, it is possible to define multiple workflows that can execute different scenarios [17].

1.3.5 The components and structure of GitHub Actions

Runners are servers that runs workflows when they're triggered. Each runner can run a single job at a time. GitHub provides Ubuntu Linux, Microsoft Windows, and macOS runners to run workflows, also is possible to host own self-hosted runners in own data center or cloud infrastructure. In yaml configuration file the type of runner, wich will be used by particular workflow, is defined by *runs-on* key.

Events in GutHub Actions trigger worflow runs, basically users can define as a event whatever thay want like a pushing new commit to a repository, opening an issue etc. In yaml configuration file event is configured by key *on*, users can define one or more events to trigger particular workflow.

Actions are custom applications for the GitHub Actions platform that performs a complex but frequently repeated task. Use an action to help reduce the amount of repetitive code that you write in your workflow files. Actions can be compared with functions in programming. For example *checkout@v3*; this action checks-out repository under the default working directory on the runner for steps, and the default location of repository when using the checkout action ($GITHUB_WORKSPACE), so workflow can access it [19]. Action *setup-java@v3* is a setup-java action that provides the following functionality for GitHub Actions runners [20]:

- Downloading and setting up a requested version of Java.
- Extracting and caching custom version of Java from a local file.
- Configuring runner for publishing using Apache Maven.
- Configuring runner for publishing using Gradle.
- Configuring runner for using GPG private key.
- Registering problem matchers for error output.
- Caching dependencies managed by Apache Maven.
- Caching dependencies managed by Gradle.
- Caching dependencies managed by sbt.
- Maven Toolchains declaration for specified JDK versions.

Jobs are a set of steps in a workflow that will be executed. Each step is shell script or action, that will be run. All jobs in configuration file are run on the same runner, so users must adhear to the same scripting language. Since all jobs are running on the one runner, they can exchanges data from one step to another. By default jobs don't have dependencies and execute in parallel with other jobs defined on particluar workflow, but in case when developer need to execute jobs in order, it can easily be done by setting job's dependencies by other jobs.

Listing 1 represents a simplified example of a GitHub Actions workflow that illustrates the described components and workflow structure. This workflow is triggered to run when someone pushes changes only to the main branch (lines 2-4). It contains just one job that consists of two steps (lines 5-11). One of these steps runs a predefined action *checkout@v4*. The second one runs a bash script that will output the string "Hello, World!" to the console. All these steps run on the "ubuntu-latest" virtual machine (line 7).

```
 1 name: GitHub Actions Demo
 2 on:
 3   push:
 4     branches: [main]
 5 jobs:
 6   Explore-GitHub-Actions:
 7     runs-on: ubuntu-latest
 8     steps:
 9       - name: Check out repository code
10         uses: actions/checkout@v4
11       - run: echo "Hello, world!"
```

Listing 1: GitHub Actions workflow example

1.3.6 Automatic token authentication

GitHub offers automatic token authentication that developers can leverage with GitHub Actions. At the commencement of each workflow job, GitHub automatically generates a unique *GITHUB_TOKEN* secret for use within the workflow. The *GITHUB_TOKEN* is employed for authentication in the workflow job, with its permissions limited to the repository containing the workflow. To use the GitHub token, you can reference it as *secrets.GITHUB_TOKEN* . When employing the repository's *GITHUB_TOKEN* for tasks, events triggered by the *GITHUB_TOKEN* do not initiate a new workflow run. This precautionary measure helps prevent inadvertent creation of recursive workflow runs [21].

1.4 GitHub Packages

GitHub packages is a platform for hosting and managing packages, including containers and other dependencies. Developers can centralize their entire software development process on GitHub, eliminating the need to publish packages on other platforms like npm, RubyGems, Apache Maven, Gradle, Docker, and NuGet. This proves advantageous for developers working on various projects, enabling them to host different software packages in a single location. Additionally, GitHub Packages allows users to host packages as public or private, providing flexibility to suit different project needs.

Combining GitHub APIs, GitHub Actions, and WebHooks allows developers to create a fully integrated end to end DevOps workflow, including CI/CD pipelines [22].

GitHub package registry allows you to view package contents, download statistics, version history to get a better understanding before you download [22].

1.4.1 Packages permissions

There are two types of packages permissions: one of them is **granular user-scoped** or **organization-scoped**, where packages permissions are scoped to a person account or organization and don't relate with repository permission, where they were published [23]. The following GitHub Packages registries support granular permissions.

- Container registry
- npm registry
- NuGet registry
- RubyGems registry

Within these registries, it is possible to release a package without associating it with a specific repository. Access to the package can then be controlled by configuring access permissions and visibility settings in the package's preferences. When a developer publishes a package, they automatically receive administrative permissions for that package. If a package is published within an organization, individuals with the owner role also gain administrative privileges for the package. Those with administrative permissions for the package have the authority to designate the package as either private or public. Moreover, they can assign access permissions specifically for the package, distinct from the permissions established at the organization and repository levels.

The second type of packages permission is **repository-scoped permission**. In this situation packages inherit permission and visibility of the repository, where they were created, so if repository is private, packages will be also private. In this case, there will be some limitations on using private packages for free. The following GitHub Packages registries only support repository-scoped permissions.

- Apache Maven registry
- Gradle registry

To use or manage a package hosted by a package registry, you must use a personal access token (classic) with the appropriate scope, and your personal account must have appropriate permissions. When you create a GitHub Actions workflow, you can use the *GITHUB_TOKEN*, described in the section 1.3.6, to publish, install, delete, and restore packages in GitHub Packages without needing to store and manage a personal access token [23].

1.4.2 Working with the Apache Maven registry

To **authenticate** to GitHub Packages, developers need to use a personal access token (classic) [24]. For Apache Maven, this authentication is accomplished by configuring the `.m2/settings.xml` file, which contains elements used to define values for Maven execution. This is achieved by adding a child <`server`> tag in the <`servers`> tag with a mapping repository defined by the <`id`> tag, a <`username`> tag set to the GitHub username, and a <`password`> tag set to the personal access token.

When authenticating to a GitHub Packages registry within a GitHub Actions workflow, the `GITHUB_TOKEN` can be used to publish packages associated with the workflow repository [24].

To **publish** Maven packages to GitHub Packages, it is necessary to configure the <`distributionManagement`> section in the Maven `pom.xml` file. Within this section, the `<repository>` tag needs to be configured with a mapping tag <`id`>, and optionally with the <`url`> tag, which is used for publishing multiple packages to the same repository. The `mvn deploy` command then initiates the deployment process, making the packages available for collaboration and use by other developers and projects within the GitHub platform. This allows for publishing several packages at once.

To **install** a GitHub package into projects, developers need to copy the XML code from the package page and add it as a dependency in the project's pom.xml file. They can then initiate the *mvn install* command, which will locally download all dependencies used in this package.

1.4.3 GitHub packages with GitHub Actions

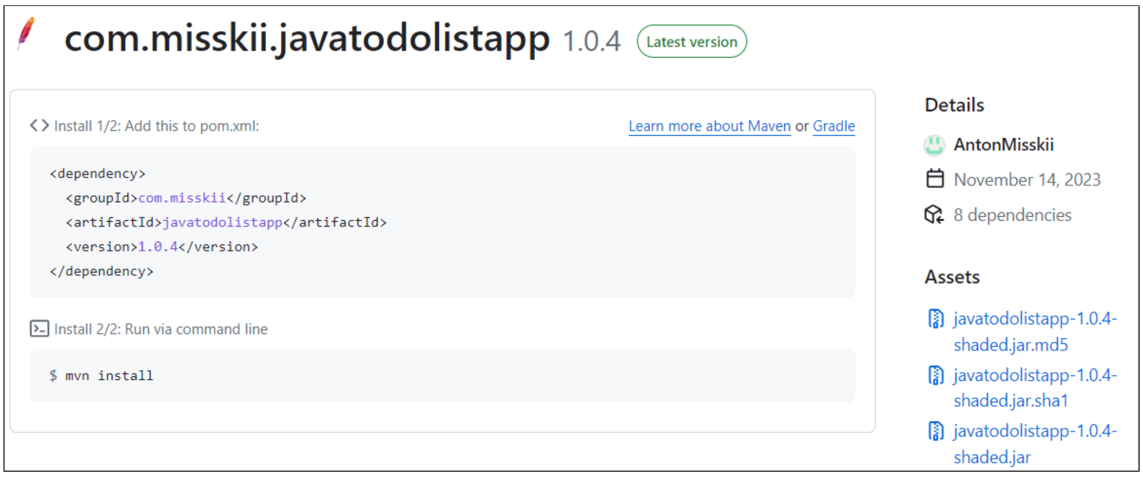
GitHub Actions provides developers with the capability to seamlessly automate the entire GitHub packages lifecycle, from publication to installation. This streamlined process becomes particularly evident when publishing new versions of applications. The workflow shown in Listing 2 is triggered when new commits are pushed to the main branch of the current repository. It comprises a single job that runs on a Windows virtual machine and executes specific steps. The workflow leverages predefined actions, specifically *checkout@v3* and *setup-java@v3*, described in the section 1.3.5. The *setup-java@v3* action modifies the *settings.xml* file, configuring the *server* tag with a *username* tag set to the environment variable *GITHUB_ACTOR* and a *password* tag set to the environment variable *GITHUB_TOKEN*. These credentials are then used by *distributionManager* in the *pom.xml* file. The *server-id* parameter is used to bind the appropriate *repository* tag inside the *distributionManager* tag with the credentials defined in *settings.xml*. Subsequently, the package is deployed and published to GitHub Packages using the commands *mvn package* and *mvn deploy*. Publishing the package also requires the *GITHUB_TOKEN* for authentication, and it is provided as a parameter.

```yaml
name: Maven Package
on:
  push:
    branches: [main]
jobs:
  build:
    runs-on: windows-latest
    permissions:
      contents: read
      packages: write
    steps:
    - uses: actions/checkout@v3
    - name: Set up JDK 11
      uses: actions/setup-java@v3
      with:
        java-version: '11'
        distribution: 'temurin'
        server-id: github # Value of the distributionManagement/repository/id field of the pom.xml
        settings-path: ${{ github.workspace }} # location for the settings.xml file
    - name: Build with Maven
      run: mvn -B package --file pom.xml
    - name: Publish to GitHub Packages Apache Maven
      run: mvn deploy -s  D:\a\todolistApp\todolistApp\settings.xml
      env:
        GITHUB_TOKEN: ${{ github.token }}
```

Listing 2: Work with GitHub Packages with GitHub Actions

1.4.4 GitHub Packages Registry

After publishing a package, it appears in the GitHub Packages Registry. Figure 1 shows the newly published package in the Github. When opening the package the user sees its name, the latest version, and in the case of Maven packages, the dependency on this package. In the "Details" section, information about the creator of this package, the creation date, and the number of dependencies included in this package are displayed. One of the key sections for this work is "Assets", where JAR files for the application can be downloaded. To track activity, there is also a download statistics for this package and previous versions. The author can provide a detailed description of the package for users who may want to use it.


com.misskii.javatodolistapp 1.0.4 Latest version

Install 1/2: Add this to pom.xml: Learn more about Maven or Gradle

```
<dependency>
  <groupId>com.misskii</groupId>
  <artifactId>javatodolistapp</artifactId>
  <version>1.0.4</version>
</dependency>
```

Install 2/2: Run via command line

```
$ mvn install
```

Details

AntonMisskii

November 14, 2023

8 dependencies

Assets

javatodolistapp-1.0.4-shaded.jar.md5

javatodolistapp-1.0.4-shaded.jar.sha1

javatodolistapp-1.0.4-shaded.jar

Fig. 1: GitHub Packages registry

2 Software license

A software license is a tool for regulating the use and distribution of a program in accordance with the rules established by the developer or vendor. The primary purpose of a software license is to protect the interests of the vendor by ensuring that the product is safeguarded from piracy, thereby protecting the investment made in creating and maintaining the project [27]. This investment is expected to be recouped when the product is used and adopted by enough users. The license typically becomes part of the application when it is released, ensuring that all terms and conditions set by the software owner are met and fully complied with.

However, when integrating software license, also protecting of user interests need to be considered. A common practice is for licensed users to receive timely program updates that add new functionality, fix bugs, or improve security. The software owner must also ensure that the license verification process for the user appears intuitive and consistent with the complexity of the licensing model [28].

2.1 Software license models

Licensing models play a key role in regulating the distribution and use of software, providing a framework for developers and businesses to determine the terms of access and use of their products. These models range from traditional approaches such as perpetual licensing, where users purchase a one-time license for ongoing use, to modern subscription-based models that offer recurring payments for ongoing access and updates.

2.1.1 Perpetual license model

The perpetual licensing model is one of the classical models for licensing software. It can use a hardware locking system or a key expiration system. Hardware locking provides several implementations, such as using a dongle – a hardware device that typically plugs into a parallel or USB port, acting as copy protection for a particular software application. Alternatively, it may rely on the Media Access Control (MAC) address of the device, a unique identifier assigned to a network interface controller [27]. However, a hardware locking system has its drawbacks. Weak node locking invites the potential misuse of purchased software, and it isn't user-friendly. It can pose problems when users want to change their device, where the software is installed, or if it's lost; consequently, they risk losing their licensed copy.

To address these disadvantages, many software vendors have developed in-house web-based systems for license distribution. These systems allow customers to log in, activate their license key, and register their product. License activation involves the user interacting with an activation wizard, which requests the entry of an activation code. This activation code is then processed by the software vendor's registration server, which returns information either allowing or denying activation [27].

2.1.2 Network-based model

Network-based models utilized by large companies to provide licensed copies to thousands of employees on their devices. In this licensing model, a company purchases a pool of a large number of licenses and then distributes copies to its employees [27]. However, this approach requires the license server to maintain constant communication with each copy of the software, which can potentially overload the server. One solution to this problem is an approach where the user temporarily acquires a licensed copy, which the license server locks temporarily and cannot provide to another user [27]. This approach has its benefits; a user with a licensed copy can work offline since constant communication with the server is not required, and it significantly reduces the load on the server.

2.1.3 Subscription model

The subscription model assumes that the user purchases a license for a certain period [27]. During this period, the user owns the software and receives all updates. Upon license expiration, the user is prompted to renew it, temporarily losing access to the application's functionality. It is primarily used by companies catering to individual users rather than large organizations and is convenient for buyers as they can choose not to renew, encouraging developers to introduce new features to retain users. It is primarily used by companies catering to individual users rather than large organizations.

2.1.4 Utility-based model

Utility-based model is also known as a license-as-needed model, where a business can acquire a pool of licenses and distribute them among its employees. Fees are charged based on the actual usage of the software as needed. This licensing model integrates well with the concurrent network-based model because the network license management system can continuously record each use and update its internal state to recognize when users have exceeded limits. The enterprise can adapt the concurrent network-based model to record each use and submit audit reports to appropriate internal teams, which in turn can submit these reports periodically to the software vendor [27].

2.1.5 Trial model

The trial licensing model provides unrestricted access to the application for users to explore its functionality by imposing limitations on the usage period or restricting the software's features. Often, solutions involve implementing both types of restrictions.

Introducing a trial version of the application is an economically beneficial approach for both the product owner and potential users. This method attracts new users by allowing them to familiarize themselves with the product before making a purchase. Users can assess the functionality, decide if the application meets their needs, and provide feedback, which is crucial for product improvement. In a sense, it is akin to beta testing, requiring no formal commitments but contributing to the enhancement and promotion of the application.

2.2 Software license implementation

When implementing a software license, it is crucial to clearly define the developer's interests it should cover and the benefits the license buyer should receive. This includes ensuring that the license provides protection against incomplete or unauthorized access to the application, facilitated through timely updates. An essential step in license implementation is selecting the appropriate licensing model that aligns with the protected product.

For instance, in the case of a basic application, creating a complex security system might be unnecessary, as it could inflate development costs and complicate the licensing process for the buyer. On the other hand, if the product's utilization is expected to generate significant revenue from licensing agreements for the vendor, implementing a more complex protection system may be advisable.

It's also vital to decide whether the application will offer a trial license period for evaluation and how trial version of the application will differ from the full version. The trial period might limit functionality, or users may not receive all new updates.

2.2.1 Proposed application license design

In this proposed solution for implementing application licensing to JavaFX application that is described in the section 3, the focus lies on restricting application functionality for users without a license and controlling the number of application license copies. Depending on the application's functionality and intent, the most suitable approach is to employ a subscription model in tandem with a trial license model.

This entails providing all interested users with a standard version of the application, but with limited features, alongside trial licenses. These trial licenses afford users the opportunity to explore the application's full suite of features initially and the option to subsequently obtain a license for a specified duration.

The implementation of these two license models is based on a market analysis of similar applications. This underscores the value of allowing buyers to trial the application and, based on their experience, opt for a subscription over a set period (e.g., 1 month, 3 months, or a year).

Figure 2 depicts a block diagram illustrating the application license validation process that takes place when the user logs into the application.

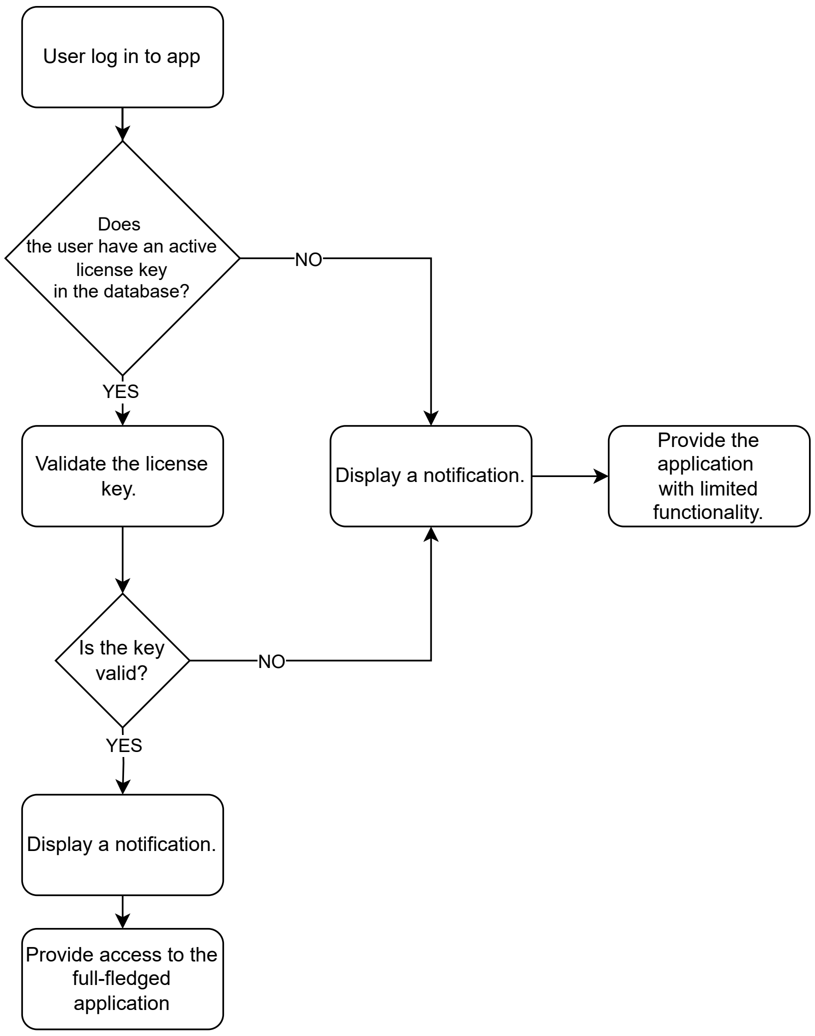

User log in to app
Does the user have an active license key in the database?
NO
YES
Validate the license key.
Display a notification.
Provide the application with limited functionality.
Is the key valid?
NO
YES
Display a notification.
Provide access to the full-fledged application

Fig. 2: Software license implementation block diagram

To implement licenses for the application, one approach is to utilize a separate license manager service. Developers or vendors can either integrate existing services with their software or develop their own service to handle license management functionalities.

In this work, a Spring RESTful API license manager has been created to provide all the necessary web services for license management. Using an external license manager has a significant advantage in security; the secret key won't be distributed with the main application, preventing crackers from writing a keygen without obtaining the private key.

2.2.2 Spring RESTful API license manager overview

The RESTful API license manager utilize two controllers designed to handle incoming HTTP requests, such as POST, GET, DELETE or PATCH and deliver appropriate responses. One of these controllers is dedicated to the web application, where administrators can manually create, delete, or modify licenses. The second controller facilitates communication between the client, which is the main JavaFX application, and the web application API. Through specific methods and URLs, this controller enables services like generating trial licenses for new users or validating license keys entered by users of the client application.

Each license key have various attributes, including the user's email (serving as the primary identifier), the license value, current status, creator information, and two timestamps indicating the creation and expiration dates to manage the license's lifespan.

To streamline communication between the main JavaFX application and the license manager, a Data Transfer Object (DTO) class is used. This DTO class contains only essential fields attributes for communication, eliminating the need for users to send HTTP requests with all license attributes, enhancing efficiency and simplicity in the communication process.

In order to strengthen security measures, the web application operates under the HTTPS protocol, safeguarding data integrity and confidentiality during communication between the client and the server.

2.2.3 Software license key generation

Through the license manager application, administrators can create new license keys and track their usage. The process of generating and validating license keys employs the asymmetric ECDSA digital signature algorithm, utilizing pre-generated private and public keys. To create a license key, the administrator enters the user's email, which is then hashed using the SHA-256 algorithm along with the license key creation date. Subsequently, the hash signed with the ECDSA private key, encoded using base-64, and the resulting encoded signature becomes the license key. All other attributes of the license key are predefined, except for the expiration date, which requires manual input by the administrator.

The diagram in Figure 3 illustrates the sequential steps involved in generating a license key for the application.

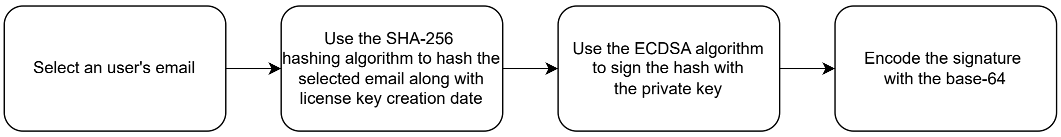

Select an user's email
Use the SHA-256 hashing algorithm to hash the selected email along with license key creation date
Use the ECDSA algorithm to sign the hash with the private key
Encode the signature with the base-64

Fig. 3: Software license key generation block diagram

The new generated license key is stored in the database with an initial status of "Unknown". This status changes dynamically when users attempt to log in to the application using the generated license key. If the method responsible for validating the license key confirms its authenticity, the status is updated to "Valid"; otherwise, it is marked as "Invalid".

2.2.4 Software license key validation

The license key validation process initiates when a user attempts to log in to the application. The client application initiates communication with the license manager application via HTTPS and sends JSON data containing the user's email and the license key.

Upon receiving the data, the license manager application retrieves the user's email. Using this email, the application searches the database for the corresponding license key creation date. It then hashes the user's email along with the creation date using the SHA-256 algorithm. Subsequently, it verifies the signature using the built-in public key. If the signature is valid, the status attribute of the license key instance is set to "Valid".

The license manager application returns information about this license key using the License Data Transfer Object (DTO) definition, which includes the updated status, to the client application. Based on the response received, the client application provides the appropriate version of the application to the user. If the license is valid, the full-featured version is provided; otherwise, only the basic features are accessible.

The process of validating a license key for the application is illustrated in Figure 4 in the accompanying diagram.

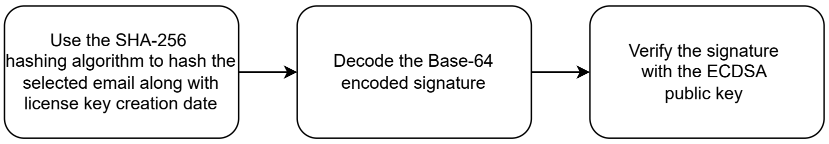

Use the SHA-256 hashing algorithm to hash the selected email along with license key creation date
Decode the Base-64 encoded signature
Verify the signature with the ECDSA public key

Fig. 4: Software license key verification block diagram

2.2.5 Software license key expiration and renewal

In this work, two licensing models are utilized: subscription and trial. They are often employed together and share similar characteristics. Both aim to familiarize users with the protected product and have expiration dates, requiring users to renew their license keys after a certain period. Therefore, it is crucial to establish a defined process for renewing users' license keys in the license manager application.

The distribution logic for trial licenses relies on one key factor: the user has never possessed the license, and there is no corresponding information about them in the license manager database. If a user attempts to reuse the trial license, the license manager returns an HTTP response message indicating that the trial license cannot be activated. Thus, renewing trial license keys is prohibited.

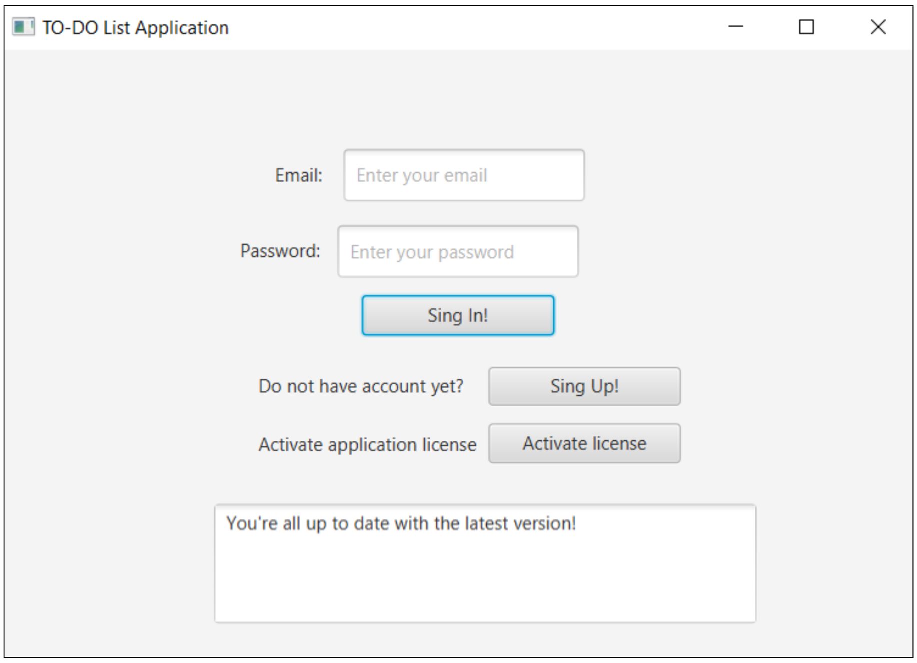
In contrast, common keys can be obtained by users, who simply input them via the license activation form. When a user attempts to log in to the JavaFX application, they send a validation request with the new license value.

3 JavaFX application

To demonstrate the process of integrating GitHub Packages into a development workflow, a Java "To-Do list" demo application was developed utilizing JavaFX and the Maven tool. The application also has access to the relational Postgresql database to save data about users and their tasks. Additionally, a robust software licensing mechanism was implemented to prevent unauthorized users from exploiting additional features.

3.1 The application overview

Figure 5 shows a login scene that is presented to the user, when application starts. It contains text fields for entering email and password, along with three buttons. The first one is the "Sing in!" button that performs login to the application. If the user's credentials are correct, the scene will transition to the main application scene; otherwise, an error message will be displayed, indicating that the credentials are incorrect. The second one is the "Sign Up!" button that will switch the login scene to the register scene. The last button "Activate license" performs the transfer to the application license scene that is described in details in the section 3.5. Additionally, the login scene includes a text area with a message dependent on the application version. If the user's version is outdated, the message informs about it, with a link to GitHub packages registry where the user can download the latest version.

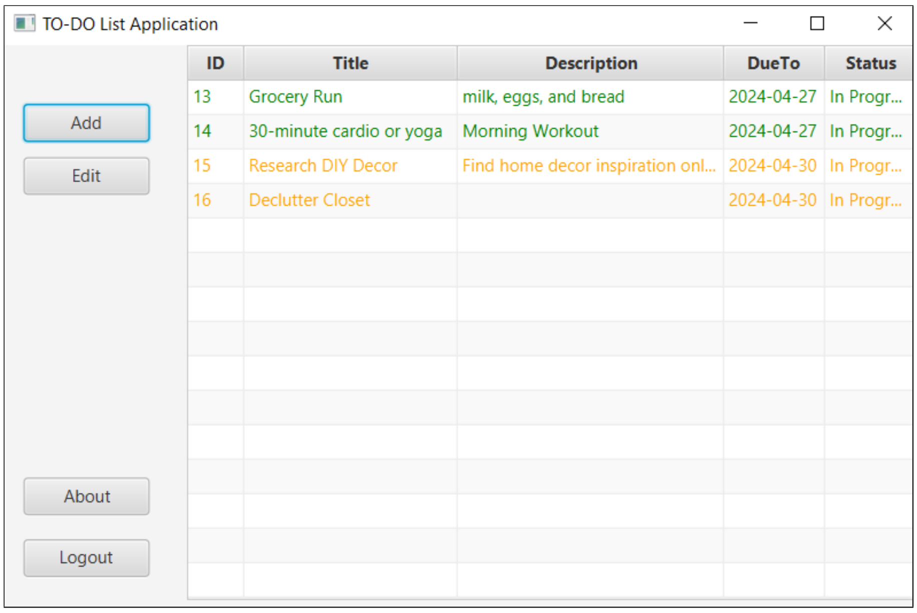


You're all up to date with the latest version!

Fig. 5: The application login scene

The main scene that is shown on Figure 6 consists of a table displaying the user's tasks, which are saved in database, accompanied by buttons to create new tasks and edit existing ones.

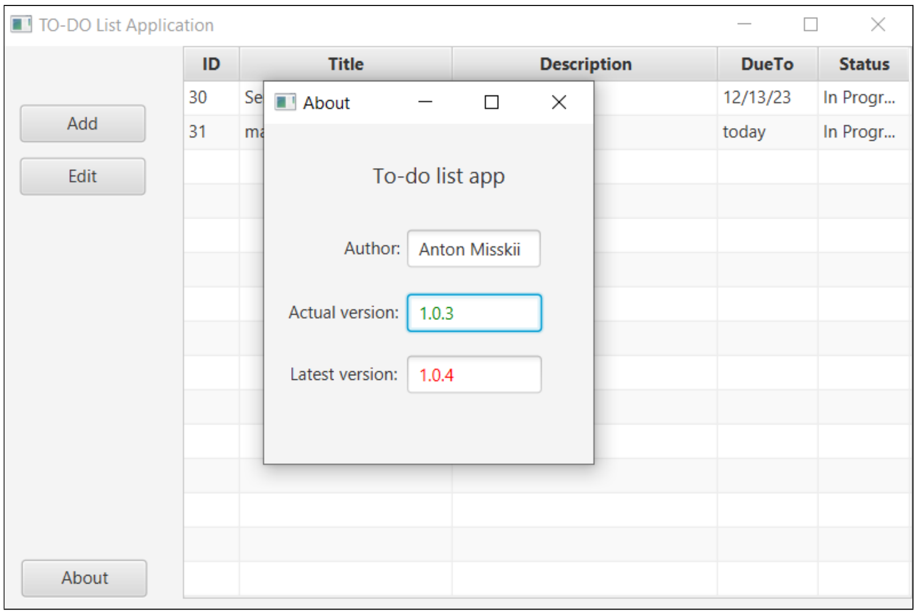
The application features a task prioritization mechanism, exclusively accessible to users with a valid application license key. Tasks are color-coded to indicate urgency, with the most critical tasks highlighted in green, the second category in blue, the third in orange, and default tasks in black text. Users can customize the priority of tasks through the edit functionality. By navigating to the edit scene, users can input the task's ID and select one of four priority buttons. Alternatively, users can specify task priorities during their creation. Users without a software license can access all other application features but will find the priority status of each task set to default.



ID	Title	Description	DueTo	Status
13	Grocery Run	milk, eggs, and bread	2024-04-27	In Progr...
14	30-minute cardio or yoga	Morning Workout	2024-04-27	In Progr...
15	Research DIY Decor	Find home decor inspiration onl...	2024-04-30	In Progr...
16	Declutter Closet		2024-04-30	In Progr...

Fig. 6: The application main scene

Additionally, there is a button that, when clicked, opens a new stage, shown on Figure 7, displaying author information and precise details about the current and latest version of the application. If the versions differ, the latest version will be displayed in red text; otherwise, it will be displayed in green text.



ID	Title	Description	DueTo	Status
30	Se		12/13/23	In Progr...
31	ma		today	In Progr...

About

To-do list app

Author: Anton Misskii

Actual version: 1.0.3

Latest version: 1.0.4

Fig. 7: The application information window

3.2 The JavaFX application structure

Applications based on JavaFX platform have hierarchically categorized components, these are stages, scenes and nodes [25].

3.2.1 Stages configuration

The top-level component in JavaFX is the **stage**, which contains all other JavaFX components. Stages serve as containers for all JavaFX objects [25]. The primary stage is automatically created by the platform, while additional stages can be created by the application. Listing 3 shows the basic configuration of the primary stage.

```java
@Override
public void start(Stage stage) throws IOException {
    Parent root = FXMLLoader.load(getClass()
    .getResource("login-page.fxml"));
    stage.setTitle("TO-DO List Application");
    stage.setScene(new Scene(root));
    stage.setResizable(true);
    stage.show();
}
```

Listing 3: Primary stage configuration

To display the primary stage, it needs to override the start method, within this method, developers can configure the stage by customizing elements such as setting the initial scene, titles, stage resizable properties, and more. The configuration is then finalized with the use of the *show* method, which needs to be called when the application starts to display the configured stage. The appearance of the stage depends on the host system and may vary between Mac OS, Windows, and Linux platforms [25].

3.2.2 Scenes configuration

Scenes are second-level elements in the JavaFX application structure, which are held by stages. Each scene consists of a scene graph, which contains various visual JavaFX elements known as Nodes. Nodes placed within the scene graph become visible components in the application [25].

One way to declare scene graph nodes in JavaFX is by using FXML files. FXML allows to describe and configure the scene graph in a declarative format. Usually with usage the FXML files developers use the Model-View-Controller (MVC) software design pattern [26]. Additionally, with FXML, it is possible to use Scene Builder, a drag-and-drop tool that provides a visual representation of the scene. Using this tool reduces the amount of code needed to write. The Scene Builder provides a clear preview of how the application will look when launched.

In JavaFX, each FXML scene file needs to have a controller, which is a Java class. The controller is specified using the *fx:controller* parameter in the FXML file and contains the code that controls the behavior of a scene that is described in the FXML file. FXML files consist of appropriate node tags with parameters, such as coordinates of each visible object, height, and width. Some of these elements, like buttons, have an *onAction* parameter, which defines the name of the function that manages the interaction with this object. In Listing 4, the code fragment for the *MainSceneController* class is demonstrated.

```java
public class MainSceneController extends GeneralController {

    @FXML
    private TableView<Task> table;
    @FXML
    private TableColumn<Task, Integer> tableId;
    @FXML
    private TableColumn<Task, String> tableTitle;
    @FXML
    private TableColumn<Task, String> tableDescription;
    @FXML
    private TableColumn<Task, Date> tableDueTo;
    @FXML
    private TableColumn<Task, String> tableStatus;
    private final TaskDAO taskDAO = new TaskDAO();
    public void fillTable() throws JsonProcessingException {
```

```
17        ObservableList<Task> tasks = taskDAO.selectAllTasksByPersonId(this.userId);
18        tableId.setCellValueFactory(new PropertyValueFactory<Task, Integer>("taskId"));
19        tableTitle.setCellValueFactory(new PropertyValueFactory<Task, String>("taskTitle"));
20        tableDescription.setCellValueFactory(new PropertyValueFactory<Task, String>("taskDescription"));
21        tableDueTo.setCellValueFactory(new PropertyValueFactory<Task, Date>("date"));
22        tableStatus.setCellValueFactory(new PropertyValueFactory<Task, String>("status"));
23        setCellFactoryBasedOnLicenseStatus(this.licenseStatus);
24        table.setItems(tasks);
25    }
```

Listing 4: Main scene controller

This class extends another class named *GeneralController* and includes the attributes *table*, *tableId*, *tableTitle*, *tableDescription*, *tableDate* and *tableStatus* all annotated with the @FXML annotation that establishes a connection between the FXML file and these Java attributes (lines 3-14). Within this class, an instance of the *TaskDAO* class is present, serving for interactions with the database (line 15).

The class contains the *fillTable* method that populates the *TableView* defined in the main scene FXML file with tasks retrieved from the database using the *taskDAO* (line 16). It sets cell value factories based on *licenseStatus* value for each *TableColumn* to specify the color of each cell with data from the Task objects.

In this way this *MainPageController* manages JavaFX scene that are originally specified in FXML file, providing seamless integration between backend logic and frontend presentation.

3.2.3 Navigating through scenes

To navigate through various scenes, it is often necessary to pass the user's identity. Listing 5 shows an example of the *switchToMainPage* function that handles these two tasks.

```
    public void switchToMainScene(ActionEvent event, int id, String status) throws IOException {
        FXMLLoader loader = new FXMLLoader(getClass().getResource("main-page.fxml"));
        root = loader.load();
        MainSceneController mainSceneController = loader.getController();
        mainSceneController.displayUser(personDAO.loginUser().get(id).getId());
```

```
 6            mainSceneController.setLicenseStatus(status);
 7            mainSceneController.fillTable();
 8            stage = (Stage)((Node)event.getSource()).getScene().getWindow()
 9            scene = new Scene(root);
10            stage.setScene(scene);
11            stage.show();
12        }
```

Listing 5: Method for transitioning between scenes

After the user logs into their account, they should only see their tasks. To achieve this, when calling the scene transition function named *switchToMainScene*, the user's ID is passed as a parameter (line 1). With this ID, the application determines which tasks belong to the logged-in user and need to be displayed. The user's ID is then retrieved from the database and stored in the *userId* variable using the *setUserId* method of the *MainSceneController* instance (line 5).

Another crucial aspect is populating the license status value from the *loginSceneController* while validating license keys. To accomplish this, the function takes the license status as a parameter and uses the *setLicenseStatus* function to propagate the license status to the main scene controller (line 6).

Following this setup, the *fillTable* function is invoked to populate the table with the corresponding data using the current user's ID.

After setting up the necessary parameters, the application proceeds with the scene change process. This process involves loading the 'main-page.fxml' FXML file from the resource directory, obtaining the *root* node of the scene graph using the *load* function (line 3). The *getController* function retrieves the associated controller (line 4), assuming the FXML file has one. A new Scene is created using the obtained *root* node, and *setScene* sets the created scene as the content for the stage, essentially specifying what should be displayed within the application window (lines 8-11).

3.3 Implementation of a Postgresql database

The primary functionality of this JavaFX application revolves around interaction with a PostgreSQL database. This interaction encompasses operations such as reading information about users, tasks, and licenses, as well as storing necessary data and providing information, such as the validation status of license keys.

The database comprises three tables: Users, Tasks, and Licenses. The relationship between the Users and Tasks tables is many-to-one, indicating that one user can have multiple tasks, with all tasks belonging exclusively to that user. Additionally, there is a one-to-one relationship between the Users and Licenses tables, as each user can have only one license key at a time.

In Figure 8, the entire database schema is shown, including all entities' columns and their data types.

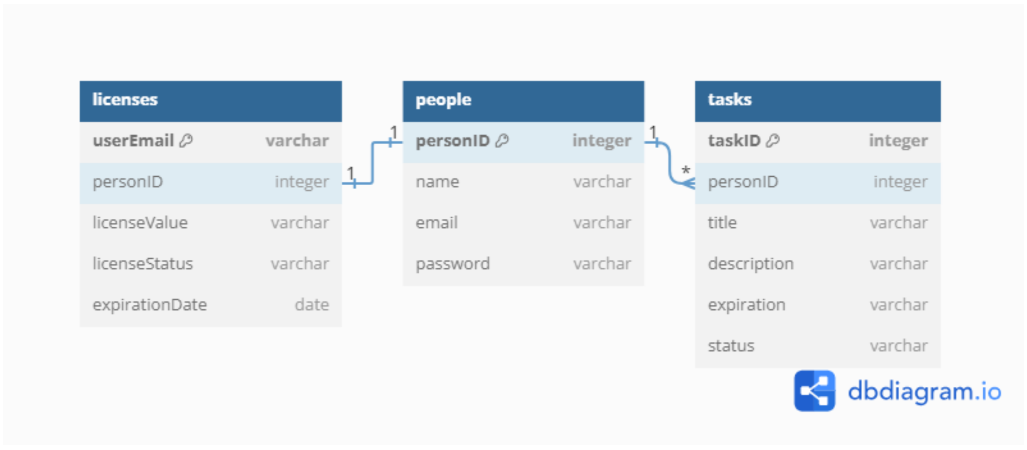


Fig. 8: Relational database schema

3.3.1 Interacting with the database

The application is utilizing JDBC API for easy access to database. JDBC is a low-level API where all actions with the database need to be implemented manually, such as translating Java objects to a string representation of a table or vice versa. This implies that all SQL queries need to be written in Java code, requiring developers to handle the intricacies of database interactions within their application logic.

Listing 6 shows an example code for interacting with the database.

```java
public class PersonDAO {
    private final Connection connection = DBUtil.getConnection();

    public void createNewPerson(Person person){
        PreparedStatement preparedStatement =
                connection.prepareStatement("INSERT INTO PERSON (
                    personName, personEmail, personPassword) VALUES (?,
                    ?, ?)", Statement.RETURN_GENERATED_KEYS);
        preparedStatement.setString(1, person.getName());
        preparedStatement.setString(2, person.getEmail());
        preparedStatement.setString(3, person.getPassword());
        preparedStatement.executeUpdate();
        ResultSet resultSet = preparedStatement.getGeneratedKeys();
        if(resultSet.next()){
            int id = resultSet.getInt(1);
        }
    }
}
```

Listing 6: Person Data Access Object class

This code represents a part of a Data Access Object (DAO) Java class designed to interact with a database. Initially, it is necessary to obtain a database connection through the *DBUtil.getConnection* function, which handles database connection details. Subsequently, the *createNewPerson* function takes a Person object as a parameter and inserts its data into the Person table using a prepared statement (lines 4-6). It sets the values for the placeholders in the SQL query using the corresponding properties of the Person object and then execute updates (lines 7-10).

3.4 The application updates

The application also includes a feature for checking version updates and displays to the user whether they have the latest version of the app or not. Listing 7 demonstrates a part of the Java class that checks the relevance of the application version.

```java
 1  public boolean compareVersions(){
 2          String apiUrl = "https://api.github.com/users/AntonMisskii/
                packages/maven/com.misskii.javatodolistapp/versions";
 3          HttpClient httpClient = HttpClients.createDefault();
 4          HttpGet httpGet = new HttpGet(apiUrl);
 5          httpGet.addHeader("Accept", "application/vnd.github+json");
 6          httpGet.addHeader("Authorization", "Bearer "+ gitToken);
 7          httpGet.addHeader("X-GitHub-Api-Version", "2022-11-28");
 8          HttpResponse response = httpClient.execute(httpGet);
 9          String responseBody = EntityUtils.toString(response.
                getEntity());
10          ObjectMapper objectMapper = new ObjectMapper();
11          JsonNode jsonNode = objectMapper.readTree(responseBody);
12          String latestVersion = jsonNode.get(0).get("name").asText();
13          setLatestVersion(latestVersion);
14          if (ACTUAL_VERSION.equals(latestVersion)) {
15              return true;
16          }
```

Listing 7: Method for checking application versions

This Java class checks versions updates by accessing a GitHub API endpoint through an HTTP GET request (lines 2-8). The GitHub API provides details on Maven package versions. The response is decoded using the *ObjectMapper* (lines 8-9) to extract the most recent version from the JSON data (lines 10-11), then *latestVersion* variable is assigned extracted value (lines 12-13). If the current version matches the latest one, the method returns true, which means that user has the latest version of application (lines 22-24).

3.5 Integration of software licensing into the application

The JavaFX demo application uses two license models: subscription and trial. The subscription model grants users access to the application's full features for a specified period. On the other hand, the trial model offers a limited-time evaluation period during which users can explore the application's functionality before deciding to get a subscription.

3.5.1 Software license activation

After the registration a user can switch to the license management scene shown in Figure 9 from the login scene.

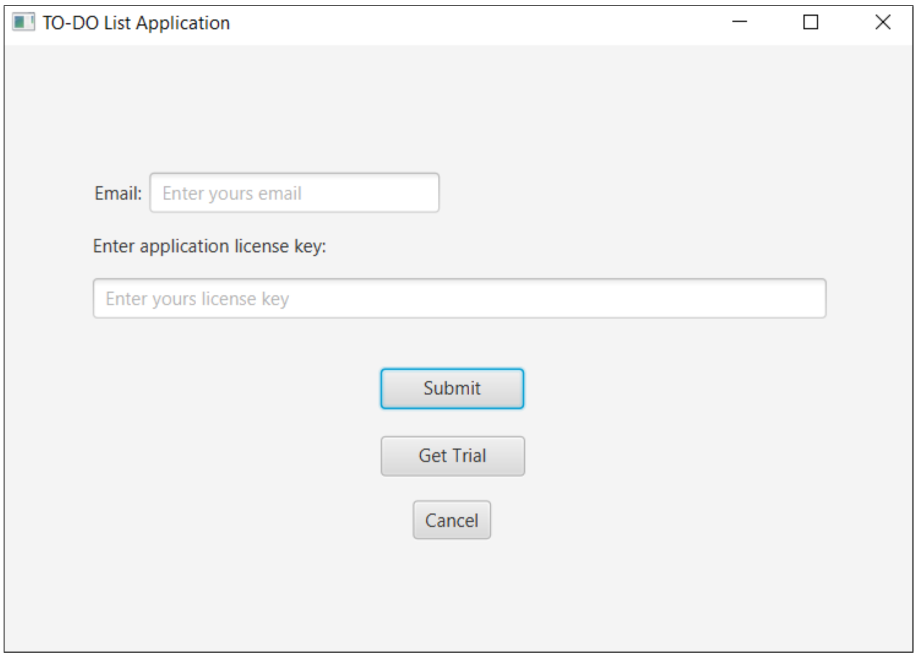


Fig. 9: The application license scene

In this scene, users will see fields to input their email and license key. Alongside, there are three buttons: "Submit", "Get Trial", and "Cancel".

Users have two options: one is to enter their email and license key obtained from the license manager application described earlier. They need to input these details correctly into the respective fields and click "Submit". After that, the application will check if the email and license key are in the correct format and save the data into the licenses table in the database, as described earlier. This data will be used later to create an HTTP POST request to the api/validate URL, directed to the license manager application, to authenticate the license key.

The second option is to get a trial license key. To get the trial license, the user should not have had an account in this application where they've already used any software key. The application will send an HTTP POST request to the api/trial URL. The license manager will verify this condition and then either send an exception or the trial license key data. This data will also be saved in the database related to the JavaFX application.

3.5.2 Communication with the license manager server

As the license manager server operates as a RESTful API application, this client application communicates with the license manager primarily through HTTP methods. The LicenseClient Java class includes methods dedicated to handling the POST requests for two main purposes: validating licenses and creating trial licenses.

The *requestTrialLicense* method depicted in the listing 8. When invoked, this method triggers a request to the license manager server's designated API endpoint responsible for generating trial license keys.

```java
    public String requestTrialLicense(String userEmail) throws
        JsonProcessingException {
        Map<String, String> jsonData = new HashMap<>();
        jsonData.put("userEmail", userEmail);
        HttpEntity<Map<String, String>> request = new HttpEntity<>(
            jsonData);
        try {
            RestTemplate restTemplate = new RestTemplate();
            String response = restTemplate.postForObject("https://
                localhost:8443/api/trial", request, String.class);
            ObjectMapper mapper = new ObjectMapper();
            JsonNode jsonNode = mapper.readTree(response);
            return jsonNode.get("licenseValue").asText();
        } catch (HttpClientErrorException.Forbidden ex) {
            String responseBody = ex.getResponseBodyAsString();
            try {
                ObjectMapper mapper = new ObjectMapper();
                JsonNode jsonNode = mapper.readTree(responseBody);
                return jsonNode.get("errorMessage").asText();
            } catch (Exception e) {
                e.printStackTrace();
            }
        } catch (JsonProcessingException e) {
            throw new RuntimeException(e);
        }
        return "";
    }
```

Listing 8: Method for requesting trial license keys

The method takes the user's email value as an input parameter. Then, a *HashMap* named *jsonData* is utilized to store the user email in key-value format (lines 2-3). Using *HttpEntity*, the method generates JSON data using the *jsonData HashMap*, incorporating the user email into it (line 4). Using the *RestTemplate* object from the Spring Web library, the method conveniently creates a POST request with the generated JSON data and sends it to the specified URL (lines 6-7). The method also parses JSON responses to extract JSON data using Jackson's *ObjectMapper* (lines 8-10).

Since creating trial licenses involves a condition, the license manager server can return two responses: if the user's email that was sent is already in the license manager database and had the license key before, the license manager server returns an *HttpClientErrorException.Forbidden* exception with the error message. The *requestTrialLicense* method can catch this exception and process it (lines 11-16). It checks the HTTP client status, and if it is forbidden, it retrieves the error message and returns it. This error message is then displayed to the user who attempted to use the trial license again or who already possessed the license key.

If the HTTP client status is OK, then the method takes the license key value from the JSON data response and saves it to the local database for subsequent authorization.

The *validateLicenseKey* method, depicted in Listing 9, follows a similar workflow to communicate with the license manager server.

```
1  public List<String> validateLicenseKey(String userEmail, String licenseKey) throws JsonProcessingException, ResourceAccessException {
2      Map<String, String> jsonData = new HashMap<>();
3      jsonData.put("userEmail", userEmail);
4      jsonData.put("licenseValue", licenseKey);
5      HttpEntity<Map<String, String>> request = new HttpEntity<>(jsonData);
6      RestTemplate restTemplate = new RestTemplate();
7      String response = restTemplate.postForObject("https://localhost:8443/api/validate", request, String.class);
8      ObjectMapper mapper = new ObjectMapper();
9      JsonNode jsonNode = mapper.readTree(response);
10     List<String> jsonResult = new ArrayList<>();
11     jsonResult.add(jsonNode.get("licenseStatus").asText());
12     jsonResult.add(jsonNode.get("expiredDate").asText());
13     return jsonResult;
14 }
```

Listing 9: Method for requesting license validation

It receives two parameters: the user email and the license key value. It creates a

HashMap named *jsonData* with these two parameters in key-value format, generates JSON from this Map using *HttpEntity*, and sends a POST request with the generated JSON data using the Spring *RestTemplate*. Then, it parses the JSON response to retrieve the license status and expiration date, adds these values to a list, and returns this list.

The *validateLicenseKey* method is called within the *switchToApp* method depicted in Listing 10 which resides in the Login scene Controller.

```
 1    public void switchToApp(ActionEvent event) throws IOException {
 2        ...
 3            if (Objects.equals(personDAO.loginUser().get(i).getEmail(),
                userEmail.getText())
 4                    && Objects.equals(personDAO.loginUser().get(i).
                        getPassword(), userPassword.getText())) {
 5                try {
 6                    List<String> licenseData = licenseClient.
                        validateLicenseKey(userEmail.getText(),
                        licenseDAO.getLicenseValueByUserID(i+1));
 7                    licenseDAO.updateLicenseStatus(licenseData.get(0),
                        LocalDateTime.parse(licenseData.get(1)),i+1);
 8                }catch (ResourceAccessException e){
 9                    if (!LocalDateTime.now().isBefore(licenseDAO.
                        getExpireDate(i+1))){
10                         licenseDAO.updateLicenseStatus("invalid", i+1);
11                    }
12                }
13                String licenseStatus = licenseDAO.getLicenseStatus(i+1)
14                if (Objects.equals(licenseStatus, "valid")){
15                    displayLicenseConfirmation("Your license is active
                        and valid");
16                }else{
17                    displayLicenseConfirmation("Your license is not
                        active or invalid");
18                }
19                switchToMainPage(event, i, licenseStatus);
20                return;
21            }
22        ...
23    }
```

Listing 10: Invoking the license validation method

If the user credentials are correct, the *validateLicenseKey* method is invoked (line 6). The returned values from this method are saved into a *licenseData* list. Subsequently, the status of the license for this user is updated in the database, along with the expiration date. Based on the returned license status, the application decides whether the user has a valid or invalid license.

Additionally, the application handles exceptions such as *ResourceAccessException* that the *validateLicenseKey* method may throw (lines 8-11). This exception occurs when the license manager server is not accessible. In such cases, if the user already has a valid license status stored locally, the application checks if the license key has not expired. If the license key is valid and has not expired, the user will still have access to the licensed version of the application, even if the server is not accessible. This ensures uninterrupted access to the application for users with valid licenses, even in scenarios where the license manager server is temporarily unreachable.

4 Java modules

This section is dedicated to the Java module system introduced in Java 9, which is considered a significant change in Java project structures, as modularizing an application affects design, compilation, packaging, and deployment processes [30].

4.1 Java environment

To better understand Java modularity and its benefits, it's crucial to understand the components of the Java environment.

There are three essential components in Java development:

- **Java Virtual Machine**, or **JVM**, is responsible for converting bytecode to machine-specific code. It is also platform-dependent and performs many functions, including memory management and security. JVM can run programs written in other programming languages that have been translated to Java bytecode [29]. JVM is contained within both the Java Development Kit (JDK) and the Java Runtime Environment (JRE).
  JVM consists of three main components or subsystems [29]:
  - **Class Loader Subsystem** is responsible for loading, linking and initializing a Java class file.
  - **Runtime Data Areas** contain method areas, PC registers, stack areas and threads.
  - **Execution Engine** contains an interpreter, compiler and garbage collection area.
- **Java Runtime Environment**, or **JRE**, is a set of software tools responsible for the execution of Java programs or applications on a system [29]. Components that are in the JRE [29]:
  - **Java Virtual Machine**: The JVM interprets Java bytecode and executes the instructions.
  - **Deployment solutions**: These simplify the activation of applications and provide advanced support for future Java updates.
  - **Development toolkits**: These are development tools designed to improve the application's user interface.
  - **Integration libraries**: These are libraries and class libraries that assist developers in creating seamless data connections between their applications and services.
  - **Language and utility libraries**: These include the Java.lang. and Java.util. packages, which are fundamental for the design of Java applications. They also handle package management.

- **Java Development Kit**, or **JDK**, is a software development kit, the JDK includes all the Java tools, executables, and binaries needed to run Java programs. This includes the Java Runtime Environment (JRE), a compiler, a debugger, an archiver, and other tools used in Java development [29]. Since Java 9, the JDK comprises approximately 90 platform modules, instead of a monolithic library, every platform module constitutes a well-defined piece of functionality of the JDK [30].

4.2 Introduction to modularity

Modularization is the process of fragmenting an application's codebase into smaller, self-contained units called modules. These modules are logically connected to each other to maintain application functionality.

4.2.1 Core Tenets of Modules

Modules must adhere to three core tenets [30]:

- **Strong encapsulation**: The module system allows separating the module code into publicly usable and internal implementation parts. This prevents other modules from accessing encapsulated code. Consequently, encapsulated code may change freely without affecting users of the module [30].
- **Well-defined interfaces**: The modules need to communicate with each other. To facilitate this, each module has its own API, which defines a portion of the module's publicly usable code. It is crucial to define a well-defined and strict API for each module as much as possible. This encapsulation ensures that changes within a module are isolated and do not affect the functioning of the rest of an application.
- **Explicit dependencies**: Modules often depend on other modules, and these dependencies are typically defined in a module description file. With explicitly defined dependencies, developers can construct a module graph. Having this module graph is important for understanding an application and ensuring that it runs with all necessary modules [30].

4.2.2 Modules declaration

Each Java module must have metadata about the module, such as module dependencies, module exports, and more. All this metadata is defined in a module declaration file named *module-info.java*, located in the *src* directory of each module. Each module declaration starts with the Java keyword *module*, followed by the module name. Modules live in a global namespace; therefore, module names must be unique [30].

After the module name, there is the main body of the module describing its characteristics. The Listing 11 illustrates the example of the module declaration file.

```java
module todolist.models {
    requires java.sql;
    exports com.misskii.todolistapp.entities;
}
```

Listing 11: Module declaration file

In this simple module declaration file, two characteristics of the *todolist.models* module are defined. The *requires* keyword indicates the dependencies used in this module. In this example, the *todolist.models* module defines just one dependency, which is the *java.sql* module. Every module implicitly requires the *java.base* platform module, as it exposes packages such as *java.lang* and *java.util*, which no other module can do without [30].

The other keyword in module's declaration files is *exports*, indicating which packages the module exports to other modules, enabling them to utilize these packages. Packages that are not exported are not accessible from other modules, even if those modules have a dependency on this module, due to the encapsulation concept of modules.

Java 9 brings new capabilities to dependency handling, introducing new ways for modules to interact and shaping the design of Java applications. Below are the key characteristics to consider when working with Java modules and their dependencies.

- The **readability** characteristic of a module allows it to access other modules. Readability is established when a module declares a dependency on another module in its declaration file. If a module attempts to use a class or interface from another module without specifying it in the *requires* clause, or if the class is not accessible, it will result in a compile-time error.
- Another important characteristic of modules is **accessibility**. When one module has set the readability to another module, it does not necessarily grant full read rights. In this situation, the normal Java accessibility rules apply: only public types in exported packages are accessible in other modules [30].
  This characteristic vividly demonstrates the strong encapsulation that comes along with Java modules and provides developers with more flexibility in code structuring. Exporting only packages meant for external use and separating them from packages meant only for internal implementation.
- When a module reads another module, it means that the module reads all dependencies of the other module transitively. To configure this behavior, **implied readability** is utilized. Implied readability is achieved by the *requires transitive* statement in the module declaration file. This approach is employed

by developers to establish transitive dependencies necessary for the module to function properly.

The Figure 10 shows a diagram of modules dependencies graph. When the *todolist.utils* module declares a dependency on the *java.sql* module using the *requires transitive* directive in its *module-info.java* file, it establishes implied readability. Consequently, any module that depends on *todolist.utils* will also transitively depend on *java.sql*.

Solid arrows are used to denote transitively used dependencies between modules, when intermittent arrows represent normal requires relationships, indicating direct dependencies between modules.

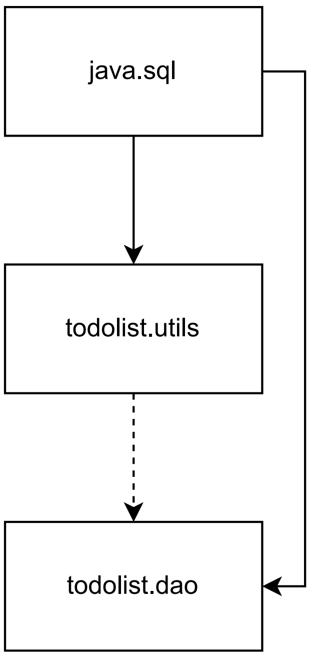


Fig. 10: Modules dependencies graph

- Sometimes modules need to be exported to other specific modules for different purposes. **Qualified exports** are used for this. The module declaration file shown in Listing 12 illustrates how to handle it.

```
module todolist.gui{
    requires javafx.fxml;
    requires javafx.controls;
    ...
    exports com.misskii.todolistapp.gui.start to javafx.graphics;
}
```

Listing 12: Qualified export example

In the *todolist.gui* module, JavaFX is used. To ensure this module works correctly, a qualified export of the package containing the class that initializes the main stage to the *javafx.graphics* module is required (line 5). This allows developers to selectively open packages to certain modules while preventing other modules from accessing them.

4.2.3 Modules resolution

The Java compiler and runtime use module descriptors to resolve the correct modules when compiling and running modules. Modules are resolved from the *module path* [30].

The *module path* serves as an alternative to the *classpath*, which was used before Java 9 to locate classes. When the JVM loads a class, it attempts to resolve its dependencies and reads the *classpath* in sequential order.

Using the classpath to locate classes presents two significant issues. Firstly, while the *classpath* might lack necessary classes, the Java application can compile without immediate errors. However, because the *classpath* is lazily loaded, issues may only arise at runtime when the JVM tries to load a required class for a specific application feature. Secondly, the *classpath* may include duplicate classes. Since the classes in the *classpath* are defined without order, the version of a class that other classes rely on may not appear, potentially leading to runtime errors.

The *module path* is designed to resolve these problems. Unlike the *classpath*, where dependencies are not explicitly defined, each module in the *module path* brings along a module declaration file. Consequently, the Java runtime and compiler know exactly, which module to resolve form the *module path* when looking for types in a given package [30].

Module resolution is the process of computing a minimal required set of modules given a dependency graph and the *root module* chosen from that graph. Every module reachable from the root module ends up in the set of resolved modules [30]. This process includes repeated phases to resolve the entire module graph [30]:

- Start with a single root module and add it to the resolved set.
- Add each required module to the resolved set.
- Repeat step 2 for each new module added to the resolved set in the step 2.

The module graph must be acyclic for the process of module resolution to conclude.

When modules cannot be resolved or conflicts arise due to different versions, addressing these issues during compilation helps ensure that the application runs smoothly without errors for users. Detecting and fixing module-related errors before deployment simplifies debugging, making it easier for developers to identify and resolve issues.

4.2.4 Modules API and Services

The Module API comprises all packages exported by a module, aimed at concealing implementation details of specific modules from other parts of the application.

Module APIs are typically implemented using standard Java interfaces, which contain function definitions, return value types, and exceptions that the functions may throw.

There are different methods for implementing a module API:

- One approach is to define the module API within the same module where the implementation classes reside. This method is suitable when the API interface is expected to have only one implementation. In this case, the module API should be located in a separate package containing only the APIs specific to that module.
- Another option is to utilize Services introduced in Java 9. This approach is suitable when the implementation classes share function signatures described in a interface located in a separate module. The module containing the implementation class requires access to both the service type and the implementation class. This approach can be advantageous when developers aim to incorporate multiple implementations of the same interface defined in the service. Such a design promotes the extensibility of the application, as introducing new implementations does not affect existing code.

An example of a module declaration file that uses the API to provide opportunities for other modules to utilize implementation instances without exporting packages is presented in Listing 13.

```
module todolist.dao {
    requires transitive todolist.utils;
    requires transitive todolist.models;
    requires javafx.base;

    exports com.misskii.todolistapp.dao.api;
    provides com.misskii.todolistapp.dao.api.PersonApi
        with com.misskii.todolistapp.dao.PersonDao;
    provides com.misskii.todolistapp.dao.api.TaskApi
        with com.misskii.todolistapp.dao.TaskDao;
    provides com.misskii.todolistapp.dao.api.LicenseApi
        with com.misskii.todolistapp.dao.LicenseDao;
}
```

Listing 13: Providing module API

The module *todolist.dao* contains the package *api* with the interfaces for each implementation class in another package. The *provides with* syntax means that this module provides API interfaces with implementation classes (lines 6-12). For example, the module provides the interface *PersonApi* with the *PersonDao* as an implementation class.

To use modules services and modules api the module need to have in the module-info.java file *uses* clause. In the Listing 14 is shown how to todolist.gui module can consume the modules api definied in the todolist.dao module.

```
module todolist.gui{
    requires javafx.fxml;
    requires javafx.controls;
    requires todolist.updater;
    requires todolist.dao;
    requires todolist.license;

    opens com.misskii.todolistapp.gui.controllers to javafx.fxml;
    exports com.misskii.todolistapp.gui.start to javafx.graphics;

    uses com.misskii.todolistapp.updater.api.UpdaterApi;
    uses com.misskii.todolistapp.dao.api.TaskApi;
    uses com.misskii.todolistapp.dao.api.PersonApi;
    uses com.misskii.todolistapp.dao.api.LicenseApi;
    uses com.misskii.todolistapp.license.api.LicenseClientApi;
}
```

Listing 14: Using modules API

The *todolist.gui* module first needs to import the packages containing the APIs it wants to use from the *todolist.dao* module. Then, it declares that it uses services provided by the *todolist.dao* module using the *uses* clause in its module-info.java file. The uses clause instruct the *ServiceLoader* that this module want to use implementations of another module. Example of service loader that is used *todolis.gui* module to load the implementation class from the *todolist.dao* module is shown in the Listing 15

```
    PersonApi personApi = ServiceLoader.load(PersonApi.class)
            .findFirst()
            .orElseThrow(() -> new RuntimeException("No implementation
                found"));
```

Listing 15: Service loader example

The *ServiceLoader* loads implementation classes lazily. This means that the implementation classes are loaded at runtime when the *ServiceLoader* is called upon to load a service provider. This lazy loading mechanism is efficient because it only loads the necessary implementation classes when needed, which can save memory and improve startup time for applications.

4.2.5 Modules evaluation

In evaluation, modules serve as a robust tool for application development, offering flexibility, understandability, and reusability [30].

Modules is versatile tool, allowing for code reuse and reliable configuration across different scenarios. The module system checks whether a specific combination of modules meets all dependencies before compiling or running the code [30]. Explicit dependencies ensure that each module operates correctly. When modules need to communicate, a well-defined module API facilitates this interaction. Additionally, encapsulation ensures that modules do not depend on implementation details, protecting against disruptions when these details change. Modular architecture improves code maintainability by enabling developers to isolate and debug issues within specific modules without affecting the entire application. It also supports scalable development by establishing explicit boundaries, allowing teams to work in parallel [30]. Moreover, it enables optimization by creating a minimal configuration of modules for distribution [30].

4.3 Migrating a JavaFX application to the module system

Before starting the application migration process to Java modules, the application structure needs to be investigated. Since modules are independent parts of an application aimed at solving specific problems, the application must be carefully examined, and an initial application migration plan must be drawn up, dividing the code into modules.

After investigating the structure of the application, the application dependencies must be analyzed. Firstly, all dependencies on which the application relies need to be defined, and then they need to be sorted to separate modules. It is also important to define internal dependencies between modules. Module declaration files also need to expose the API of each module. It is important to set appropriate APIs for each module to ensure that implementation details are concealed, and simultaneously, that modules export everything that might be needed to use them fully.

The module diagram of the JavaFX demo application shown in Figure 11 indicates the relationships of the internal application modules; solid arrows represent transitive dependencies, indicating that a module depends on another module transitively through other modules, while intermittent arrows represent normal dependencies, indicating direct dependencies between modules.

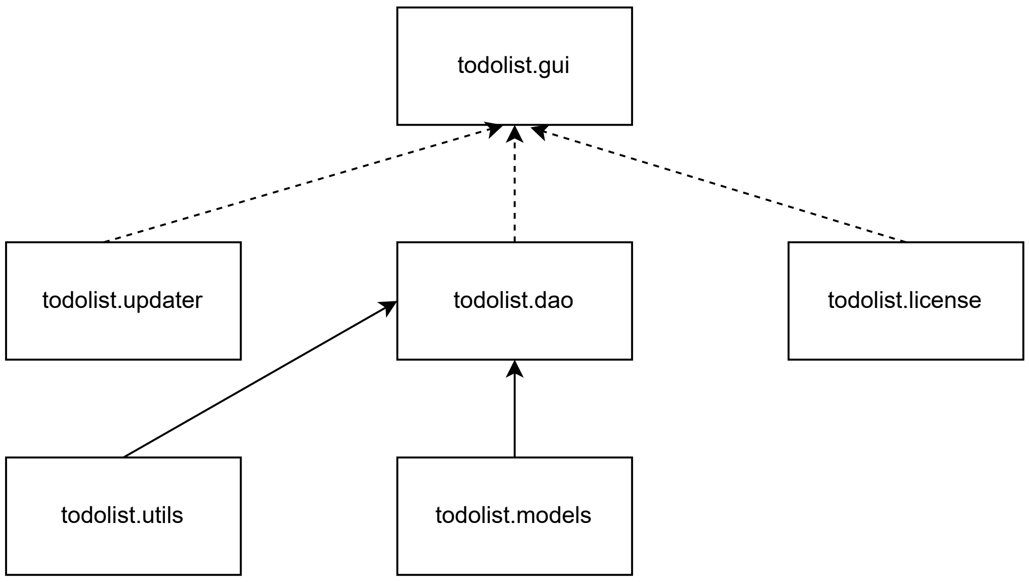


Fig. 11: The modular JavaFX demo application diagram

The modular JavaFX application consists of six modules. Each module is dedicated to specific application functionalities. For instance, the *todolist.updater* module verifies if the user is using the latest application version, the *todolist.dao* module manages interactions with the database, the *todolist.license* module handles JavaFX demo application license keys, the *todolist.models* module defines classes describing license, users, and tasks objects, and the *todolist.utils* modules manage database connections.

The primary module, *todolist.gui*, hosts the Java class initiating the application and encompasses all FXML views of scenes and their controllers. The declaration file of *todolist.gui* module, that is described in the 4.2.4 Section. According to this file, *todolist.gui* not only requires platform modules but also external modules such as *javafx*, and internal modules including *todolist.updater*, *todolist.dao*, and *todolist.license*. It utilizes their APIs to load their implementation classes with *ServiceLoader* at runtime.

4.3.1 Modules with Apache Maven

As the original demo application was developed using the Apache Maven tool, migrating it to Java modules requires restructuring the application. The original pom file needs to be split into several smaller pom files, each belonging to a particular module and defining only their dependencies. Additionally, a parent pom file needs to be created, which contains the entire list of internal application modules and defines generic dependencies or plugins.

4.4 Custom runtime images

In the Java Platform Modules System, developers can create minimal custom runtime images with the `jlink` tool that links the modules. Linking is the process of bringing together compiled artifacts into an efficiently executable form [30].

Creating a custom runtime image is beneficial for several reasons [30]:

- The `jlink` tool creates an application distribution and JVM, ready to be shipped.
- The size of this runtime image will be lower as only the modules that the application uses are linked into the runtime image.
- As the custom runtime image contains only modules that the application uses, developers may control them to exclude potentially vulnerable modules.
- A custom runtime image is fully self-contained. It bundles the application modules with the JVM and everything else it needs to execute the application. No other Java installation (JDK/JRE) is necessary [30].

4.4.1 Automatic modules and jdeps tool

Before creating a custom runtime image, it's crucial to investigate the application's dependencies. Many libraries have not modularized yet and are represented in modular applications as automatic modules. Automatic modules lack a module-info.class module descriptor, they require transitive dependencies on all other application modules and export all their packages.

The issue with automatic modules arises because they lack a module description file, and thus, *jlink* cannot resolve and link them with other modules.

The solution is to either use already modularized libraries or create module descriptor files for these modules. This can be achieved using the *jdeps* tool, which generates module descriptor files for JARs. Developers need to resolve all module dependencies and recursively create module descriptors for them if they do not already exist. Once the module descriptor files are created, the next step is to compile them. This can be done using the *javac* tool. After compiling the module descriptors, it is necessary to add them to the JAR files of libraries.

The example workflow is shown in Listing 16, it demonstrates steps to add module-info.java file to *postgresql* automatic module jar.

Listing 16: Add module descriptors to automatic modules

```
1) jdeps --generate-module-info . checker-qual.jar
2) javac --patch-module \
org.checkerframework.checker.qual=checker-qual.jar \
module-info.java
```

```
3) jar uf checker-qual.jar -C . module-info.class
4) jdeps --module-path . --add-modules \
org.checkerframework.checker.qual \
--generate-module-info . postgresql.jar
```

The first command uses the *checker-qual.jar* file to generate a module-info.java file uses *–generate-module-info* option. The second command compiles the module-info.java file using the *javac* compiler. The *–patch-module* option specifies that the *checker-qual.jar* file should serve as a patch for the `checker.qual` module. The third command updates the *checker-qual.jar* file by adding the module-info.class file to it. The last command analyzes the dependencies of the *postgresql.jar* file. It specifies the module path and adds the compiled *checker.qual* module. Eventually generating a module-info.java file for the *postgresql.jar* library. Finally, the new module-info file for *postgresql.jar* must be compiled and integrated into the library JAR using *javac* and *jar* tools, following a similar process as with the *checker-qual* library.

4.4.2 Creating custom runtime images

After ensuring that the application does not have automatic modules, developers can create a custom runtime image. The *jlink* tool is used to create this image.

Listing 17: Custom runtime image creation

```
jlink --module-path . --add-modules todolist.dao, \
todolist.gui, todolist.license, \
todolist.models,todolist.updater, \
todolist.utils,org.postgresql.jdbc \
--launcher todolistapp=todolist.gui --output todolist-image
```

JLink specifies the module path where all necessary modules are located, including the application modules to be added to the custom image, including the *org.postgresql.jdbc* that is created from an automatic module. Define the name of a launcher script, indicating the module it should run, and set the output directory where the image is generated.

After this, the new directory with the custom image is created. This image includes only the modules that the application needs and can be distributed.

Conclusion

One of the main tasks of this work is to study and implement the GitHub Packages service into the development and distribution process of Java applications. Since this service cannot be considered in isolation, it is being used in conjunction with other services and tools such as Maven, Git, GitHub, and GitHub Actions, all of which are discussed in Chapter 1.

Another crucial aspect being addressed in this work is licensing. Chapter 2 delves into different licensing models, proposing a merger of the trial period model and the subscription model to align with project requirements. To manage application license keys effectively, a dedicated web application is being developed. This application allows the administrator to create keys manually or via its API, facilitating tasks such as generating trial licenses, validating license keys, and distributing them to users. The Chapter 2 also elaborates on the operations involved in key expiration and renewal, essential for the licensing models explored.

A comprehensive demo application has been selected to showcase various functionalities, aiding in understanding real-world development and maintenance challenges. Chapter 3 provides a detailed overview of the JavaFX application's structure and its interaction with external services. It discusses processes like fetching the latest version from GitHub Packages via HTTP requests and communication with the license manager to activate licensed versions. Upon updating the application and pushing new code to GitHub, an automatic GitHub Actions workflow is initiated. This workflow handles all necessary steps, including building and publishing application artifacts to the GitHub Packages service. Users can then download the ready-made JAR file from there to begin using the application.

Chapter 4 provides a detailed exploration of Java modules and their benefits in the application development process. Building upon the JavaFX demo application outlined in Chapter 3, this chapter includes an example of migrating a standard Java application to a modular one. It offers a comprehensive overview of the application structure and describes the use of custom runtime images, which can facilitate the distribution of modular applications.

Bibliography

[1] GitHub Docs. *Introduction to GitHub Packages*. [online] Available at: https://docs.github.com/en/packages/learn-github-packages/introduction-to-github-packages. [Accessed: 2023-11-2].

[2] Apache Maven. *Apache Maven - What is Maven?*. [online] Available at: https://maven.apache.org/what-is-maven.html. [Accessed: 2023-10-20].

[3] Apache Maven. *Introduction to the POM - Apache Maven.* [online] Available at: https://maven.apache.org/guides/introduction/introduction-to-the-pom.html#what-is-a-pom. [Accessed: 2023-10-20].

[4] Apache Maven. *Maven Features - Apache Maven.* [online] Available at: https://maven.apache.org/maven-features.html. [Accessed: 2023-10-20].

[5] Apache Maven. *Introduction to the Maven Lifecycle - Apache Maven.* [online] Available at: https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html[Accessed: 2023-10-20].

[6] Akhtar, H. *Understanding Maven Goals and Plugins - BrowserStack Guide.* BrowserStack. [online] Available at: https://www.browserstack.com/guide/maven-lifecycle#:~:text=Management%20with%20Selenium-,Understanding%20Maven%20Goals%20and%20Plugins,-In%20Maven%2C%20goals. [Accessed: 2023-10-20].

[7] Chacon, S., Straub, B. (2014). *Pro Git.* 2st ed. New York: Apress. ISBN 978-1484200773.

[8] GNU RCS. *GNU Revision Control System.* [online] Available at: https://www.gnu.org/software/rcs/. [Accessed: 2023-10-22].

[9] Lionetti, G. *Version Control: Centralized vs. DVCS.* Atlassian. [online] Available at: https://www.atlassian.com/blog/software-teams/version-control-centralized-dvcs. [Accessed: 2023-10-22].

[10] Git SCM. *Git - The Staging Area.* [online] Available at: https://git-scm.com/about/staging-area. [Accessed: 2023-10-22].

[11] Git SCM. *Git - Distributed Git.* [online] Available at: https://git-scm.com/about/distributed. [Accessed: 2023-10-24].

[12] Lutkevich, B, Courtemanche, M. *GitHub Definition.* TechTarget. [online] Available at: https://www.techtarget.com/searchitoperations/definition/GitHub. [Accessed 2023-10-27].

[13] Lionneti, G. *Version Control: Centralized vs. Distributed.* Atlassian. [online] Available at: `https://www.atlassian.com/blog/software-teams/version-control-centralized-dvcs` [Accessed 2023-10-28].

[14] Dabbish, L., Stuart, C., Tsay, J., and Herbsleb, J., 2012. *Social coding in GitHub: transparency and collaboration in an open software repository.* In: *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work, Seattle, Washington, USA*. New York: Association for Computing Machinery. DOI: `10.1145/2145204.2145396`.

[15] GitHub Docs. *About Pull Requests.* [online] Available at: `https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests`. [Accessed 2023-11-27].

[16] Kinsman, T., Wessel, M., Gerosa, M. A., and Treude, C., 2021. *How do software developers use GitHub Actions to automate their workflows?.* In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. DOI: `10.1109/MSR52588.2021.00054`.

[17] Mastropaolo, A., Zampetti, F., Bavota, G., and Di Penta, M., 2024. *Toward automatically completing GitHub workflows.* In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, Lisbon, Portugal.* New York: Association for Computing Machinery. DOI: `10.1145/3597503.3623351`.

[18] Gousios, G., Storey, M.-A., and Bacchelli, A., 2016. *Work practices and challenges in pull-based development: the contributor's perspective.* In: *Proceedings of the 38th International Conference on Software Engineering, Austin, Texas.* New York: Association for Computing Machinery. DOI: `10.1145/2884781.2884826`.

[19] GitHub Actions. *Checkout repository.* [online] Available at: `https://github.com/actions/checkout` [Accessed 2023-11-6].

[20] GitHub Actions. *Setup Java JDK.* [online] Available at: `https://github.com/marketplace/actions/setup-java-jdk` [Accessed 2023-11-6].

[21] GitHub Docs. *Automatic token authentication.* [online] Available at: `https://docs.github.com/en/actions/security-guides/automatic-token-authentication` [Accessed 2023-11-4].

[22] Dulanga, C., 2020. *GitHub Package Registry: Is it worth trying out?*. [online] Available at: `https://blog.bitsrc.io/github-package-registry-is-it-worth-trying-out-62163aa3d518` [Accessed 2023-11-5].

[23] GitHub Docs. *About permissions for GitHub Packages*. [online] Available at: `https://docs.github.com/en/packages/learn-github-packages/about-permissions-for-github-packages` [Accessed 2023-11-6].

[24] GitHub Docs. *Working with the Apache Maven registry*. [online] Available at: `https://docs.github.com/en/packages/working-with-a-github-packages-registry/working-with-the-apache-maven-registry` [Accessed 2023-11-6].

[25] Java Developer. *JavaFX Application Basic Structure By Example*. [online] Available at: `https://dev.java/learn/javafx/structure/` [Accessed 2023-11-3].

[26] Java Developer. *Using FXML*. [online] Available at: `https://dev.java/learn/javafx/fxml/` [Accessed 2023-11-3].

[27] Ferrante, D., 2006. *Software Licensing Models: What's Out There?*. In: *IT Professional*, vol. 8, no. 6. DOI: `10.1109/MITP.2006.147`.

[28] Noorian L., Perry M., 2009. *Autonomic Software License Management System: An Implementation of Licensing Patterns*. In: *Proceedings of the 2009 Fifth International Conference on Autonomic and Autonomous Systems*. DOI: `10.1109/ICAS.2009.15`.

[29] IBM Cloud Education, 2021. *JVM vs. JRE vs. JDK: What's the Difference?*. [online] Available at: `https://www.ibm.com/blog/jvm-vs-jre-vs-jdk/`. [Accessed 2024-4-20]

[30] Mak, S., & Bakker, P., 2017. *Java 9 Modularity: Patterns and Practices for Developing Maintainable Applications*. 1st ed. Sebastopol, Calif: O'Reilly Media. ISBN 978-1491954164.

Symbols and abbreviations

POM	Project Object Model
JAR	Java Archive
POM	Project Object Model
VCS	Version Control System
LVCS	Local Version Control System
RCS	Revision Control System
CVCS	Centralized Version Control System
DVCS	Distributed Version Control System
SaaS	Software as a Service
CI/CD	Continuous Integration/Continuous Deployment or Continuous Delivery
MAC	Media Access Control
REST	Representational State Transfer
API	Application Programming Interface
DTO	Data Transfer Object
ECDSA	Elliptic Curve Digital Signature Algorithm
SHA-256	Secure Hash Algorithm 256-bit
OS	Operating System
MVC	Model-View-Controller
DAO	Data Access Object
SQL	Structured Query Language
JVM	Java Virtual Machine
JRE	Java Runtime Environment
JDK	Java Development Kit

A Structure of the archive with the source files

- `sorce-files-BP-Misskii` — Root directory
  - `todo-application.zip` — The archive with the demo JavaFX application source code
    - `src` — The directory with the application source code
      - `main`
        - `resources` — The directory with the scenes defenition fxml files
        - `java`
          - `com.misskii.javatodolistapp` — The root package of the application
            - `util` — Package containing utility classes
            - `updater` — Package containing classes responsible for updating application
            - `models` — Package containing entity classes
            - `license` — Package containing classes related to licensing of the application
            - `dao` — Package containing classes responsible for interacting with the database
            - `controllers` — Package containing controller classes for fxml scenes
            - `Main.java` — Main entry point class of the application
            - `Application.java` — Class responsible for application configuration and setup
          - `module-info.java` — Module declaration file
    - `.mvn`
    - `.idea`
    - `.github`
      - `workflows`
        - `maven-publish.yaml` — The GitHub Actions workflow definition file
    - `.git`
    - `pom.xml` — File used by Maven to configure the project
    - `mvnw.cmd`
    - `mvnw`
    - `.gitignore`
  - `license-manager.zip` — The archive with the license manager application source code

B Manual for the Applications

The source code files for the demo JavaFX To-do list and license manager applications are included in the archive attached to this thesis.

1 The Demo JavaFX Application

1.1 Start the Demo JavaFX Application

The entire source code for the ToDo List application is available on GitHub. To clone the repository to your local machine, use the following command:

```
git clone https://github.com/amisskii1/todolistApp
```

Then, run the application.

Alternatively, users can download a directly runnable JAR file from the GitHub packages dedicated to this repository. The shaded JAR file can be found in the assets section. Visit the following link to download the JAR file:

`https://github.com/amisskii1/todolistApp/packages/1962002`

The application uses a GitHub token to communicate with the GitHub API. Users need to create a new environment variable with the GitHub token for the application to work correctly. Additionally, ensure that Java JDK is pre-installed on your machine.

If users want to start the modular JavaFX application, the source code is available in a separate GitHub repository:

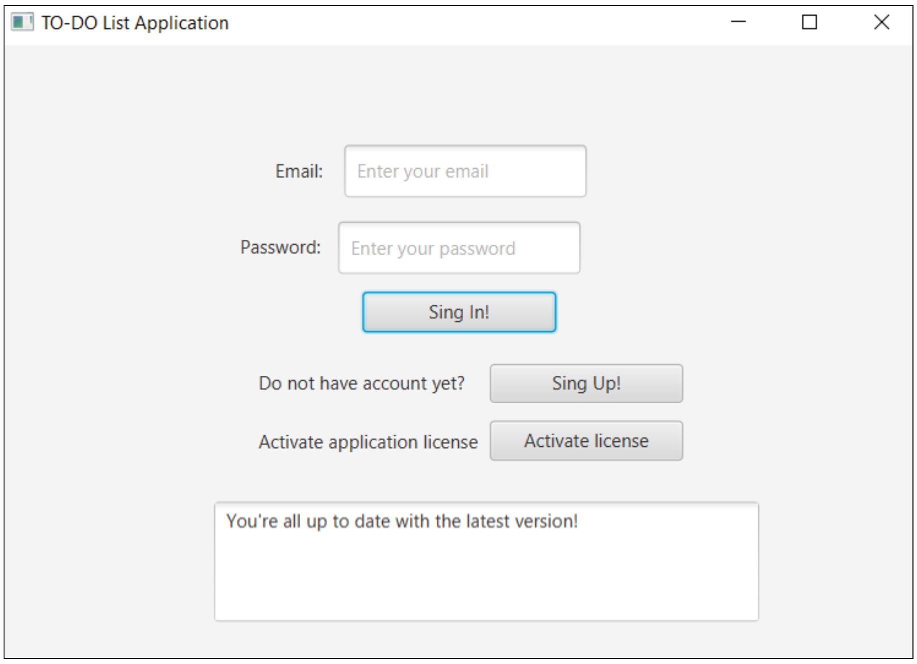
`https://github.com/amisskii1/modular-application`

To run the modular application, go to the releases section of the repository and choose one of the latest releases. This will allow you to download a custom image that already contains the binary file necessary to start the application.

1.2 Using the JavaFX Application

After the application starts, the user sees the login page. Here, the user can enter their credentials to access the application if they already have an account. Alternatively, the user can register for a new account on the registration page. The user also has the option to activate the software license on the activation license page. On this page, the user can enter their email and license key if they have one, or simply enter their email to receive a trial license key if they do not already have one. If the user has a valid license key and logs into the application, the application will inform them with a new window.

On the main application page, the user will see a table with tasks. On this page, the user can navigate to the form for creating a new task or editing an existing task. When creating a new task or editing one, the user can select the task's priority, and the task will be colored appropriately based on its priority. This feature is only available for users with license keys. On the main page, the user can also navigate to the "About" window, where information about the current and latest versions is displayed.



You're all up to date with the latest version!

Fig. B.1: The ToDo list application login scene

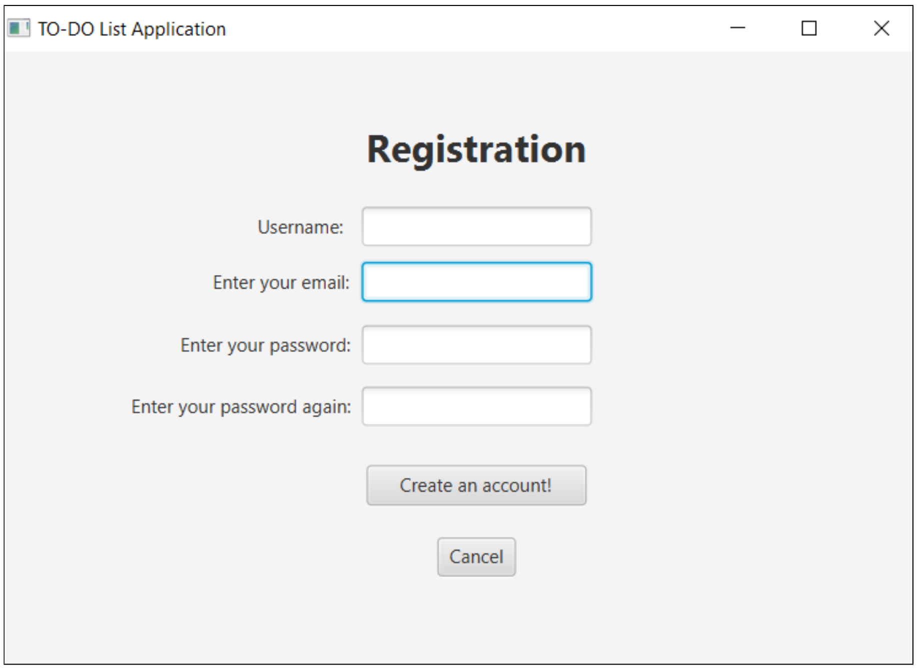


Fig. B.2: The ToDo list application register scene

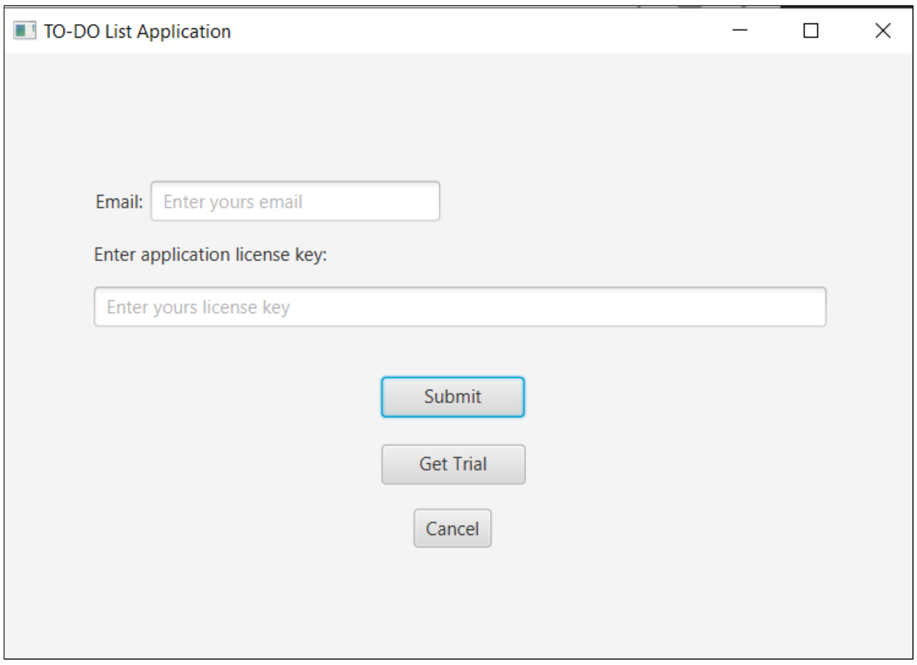
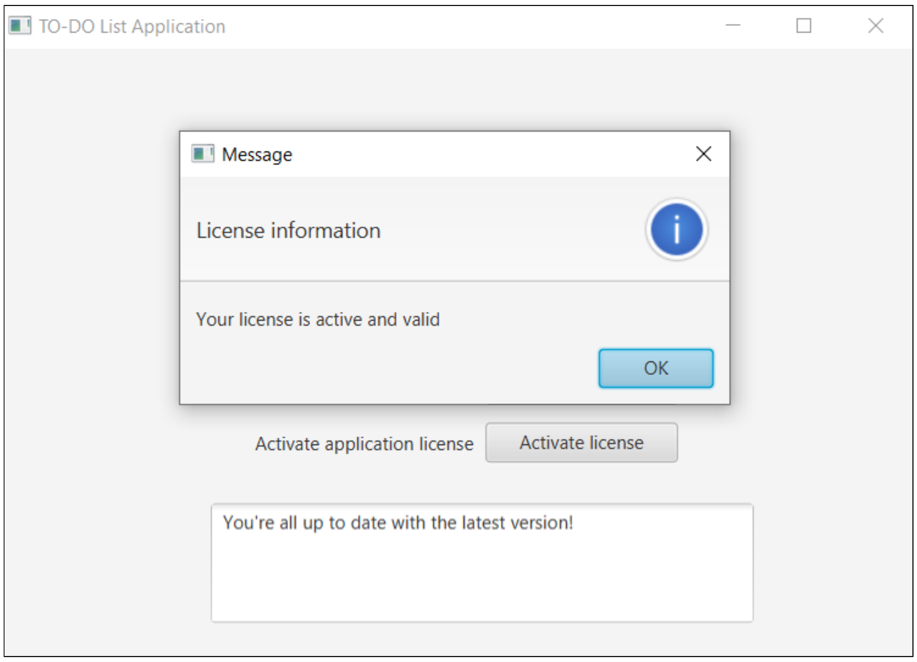


Fig. B.3: The ToDo list application license scene


TO-DO List Application

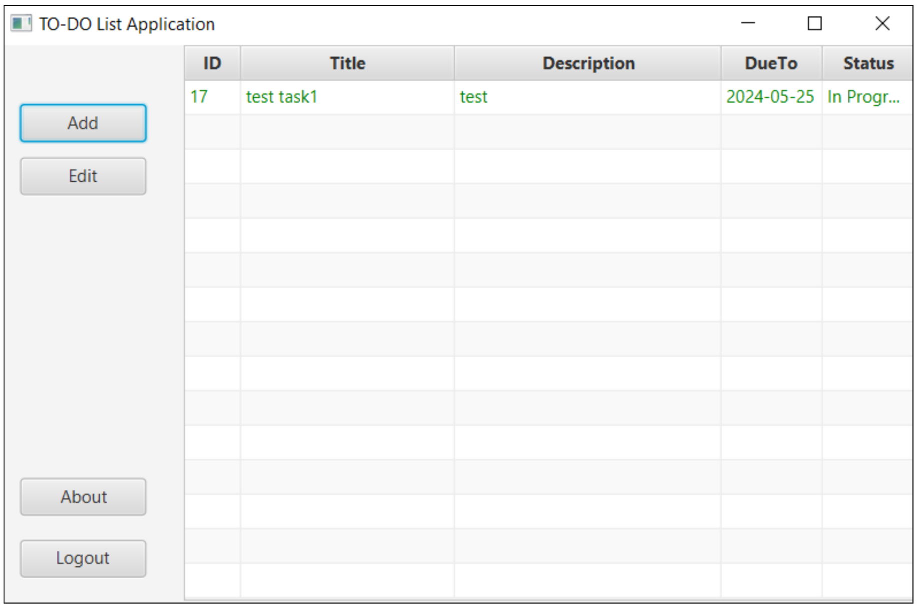
Message

License information

Your license is active and valid

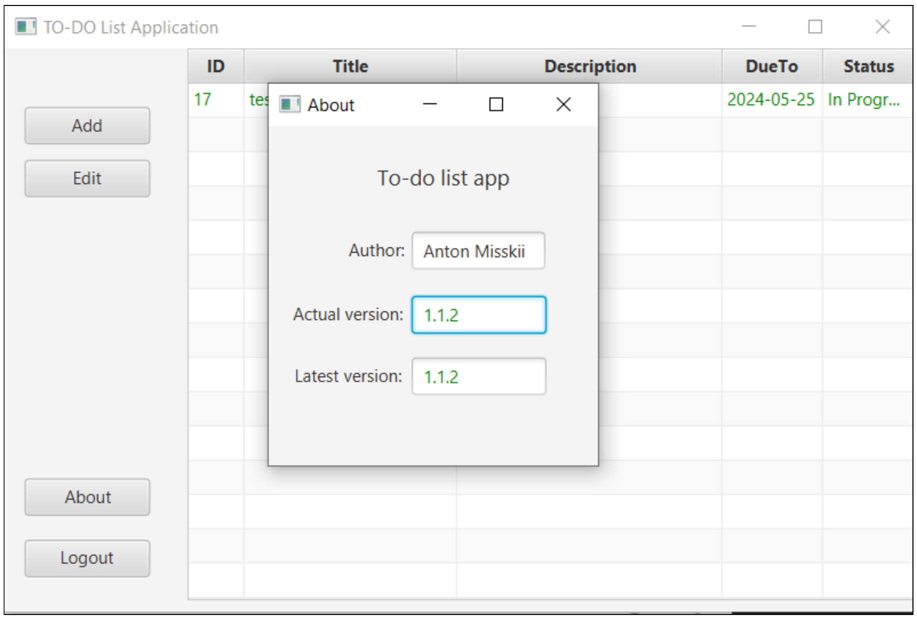
You're all up to date with the latest version!

Fig. B.4: The ToDo list application license information stage


TO-DO List Application

ID	Title	Description	DueTo	Status
17	test task1	test	2024-05-25	In Progr...

Fig. B.5: The ToDo list application main scene



To-do list app

Author: Anton Misskii

Actual version: 1.1.2

Latest version: 1.1.2

Fig. B.6: The ToDo list application about stage

2 The License Manager Application

2.1 Start the License Manager Application

The source code for the license manager application is available on GitHub. To clone the repository to your local machine, use the following command:

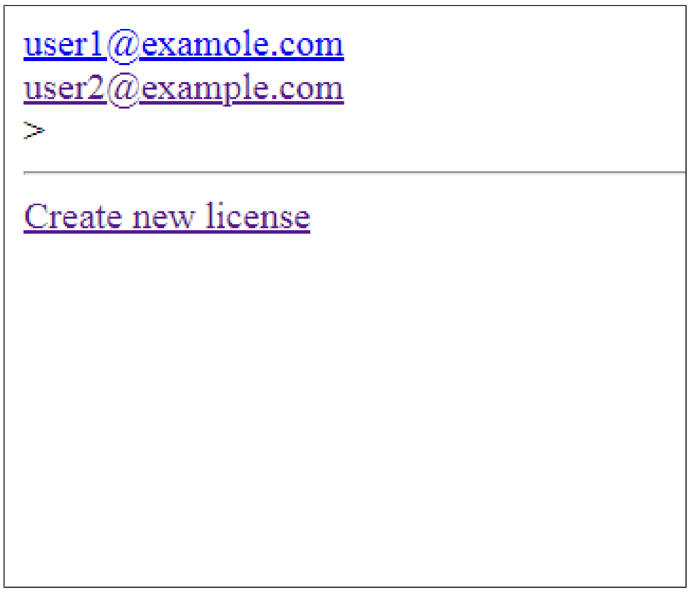
```
git clone https://github.com/amisskii1/license-manager
```

Then, run the application. By default, the application runs on `localhost` with port number 8443, indicating that the application uses the HTTPS security protocol.

2.2 Using the License Manager Application

The `https://localhost:8443/licenses` is the main application page where all users' emails who have or had a license key are displayed. From this page, the admin can navigate to the creation of a new license key page at `https://localhost:8443/licenses/new` and enter the user's email for whom the admin wants to generate a new key. Then the admin will be automatically redirected to the creation form where the default values are predefined, such as the license key value, the user's email, license status, created by, and date of creation. On this form, the administrator can only configure the expiration date. After the new license key is created, the admin will be redirected to the main page.

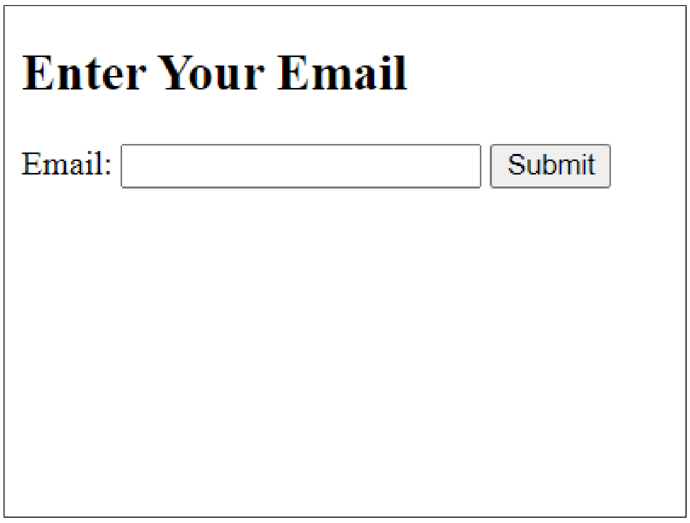
The admin can click on every record to see license key details. On this page, the admin can edit each field or delete the record with the license key.



user1@examole.com
user2@example.com
>

Create new license

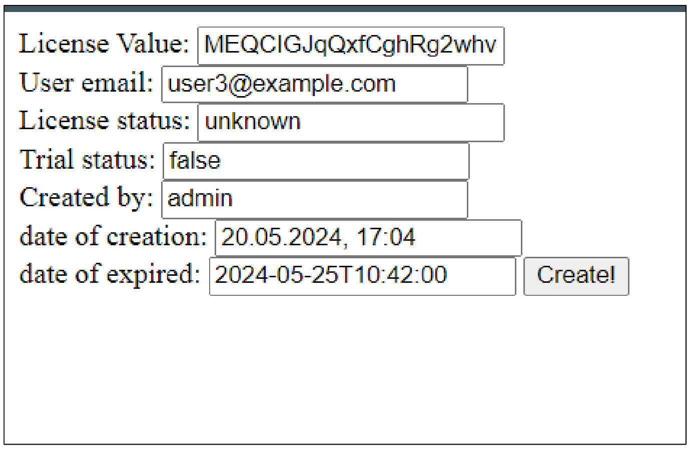
Fig. B.7: The license manager application licenses page



Enter Your Email

Email: Submit

Fig. B.8: The license manager application email form



License Value: MEQCIGJqQxfCghRg2whv

User email: user3@example.com

License status: unknown

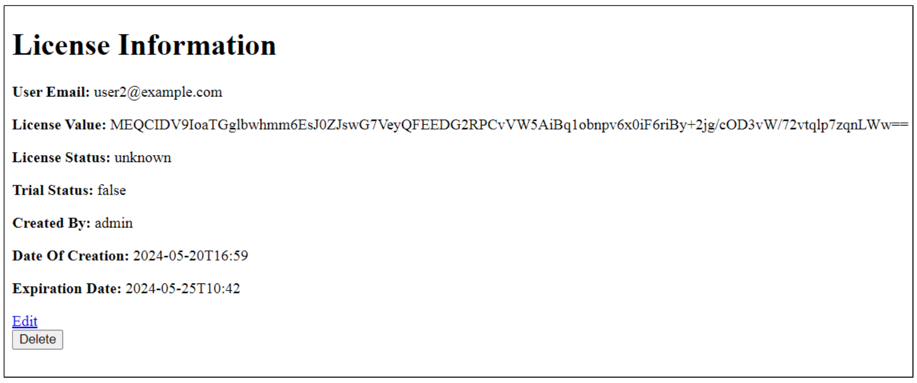
Trial status: false

Created by: admin

date of creation: 20.05.2024, 17:04

date of expired: 2024-05-25T10:42:00

Fig. B.9: The license manager application new license creation form



License Information

User Email: user2@example.com

License Value: MEQCIDV9IoaTGglbwhmm6EsJ0ZJswG7VeyQFEEDG2RPCvVW5AiBq1obnpv6x0iF6riBy+2jg/cOD3vW/72vtqlp7zqnLWw==

License Status: unknown

Trial Status: false

Created By: admin

Date Of Creation: 2024-05-20T16:59

Expiration Date: 2024-05-25T10:42

Fig. B.10: The license manager application license key information page