Bachelor Project

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computers

Web application for e-mail management

Valeriia Chekanova

Supervisor: Ing. Jan Zídek
Field of study: Software Engineering and Technology
May 2022

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Chekanova Valeriia** Personal ID number: **492197**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Software Engineering and Technology**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Web application for e-mail management

Bachelor's thesis title in Czech:

Webová služba na zpracování emailů

Guidelines:

This bachelor thesis aims to create a web service for e-mail management.
Guidelines:
1) Become familiar with e-mail processing (protocols, agents, security side).
2) Analyze the existing method of communication between services.
3) Define the functional, non-functional requirements, and acceptance criteria.
4) Design the scalable application. The application should support user roles distinction, allow e-mail sending, storing in the database, different e-mail configuration and browsing sent e-mails. Implement a health check module.
5) Implement and deploy the back-end part of the application.
6) Create test scenarios and test the application in the test environment. Cover code with unit tests.

Bibliography / sources:

[1] D. Crocker. Internet Mail Architecture. 2009
[2] Yasushi Saito, et al. Manageability, availability and performance in Porcupine: a highly scalable, cluster-based mail service. 1999
[3] Nick Christenson, et al. Highly Scalable Electronic Mail Service Using Open Systems. 1997
[4] Bruce Schneier. E-Mail Security: How To Keep Your Electronic Messages Private. 1995
[5] Evgen Verzun. The Art of Email Security: Putting Cybersecurity In Simple Terms. 2020

Name and workplace of bachelor's thesis supervisor:

Ing. Jan Zídek Centrum znalostního managementu

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **12.01.2022** Deadline for bachelor thesis submission: _____

Assignment valid until: **30.09.2023**

Ing. Jan Zídek Supervisor's signature	Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce her thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt	Student's signature

Acknowledgements

I would like to acknowledge my supervisor Ing. Jan Zídek for his patient guidance and valuable advice. Furthermore, I would like to thank my parents for supporting me throughout the years. Many thanks also go to Ph.D. Andrey Bondarev for his contagious optimism that has helped me to finish this thesis.

Declaration

I declare that this text presents my own work, and I have quoted all relevant sources of information used.

Prague, 19. May 2022

Valeriia Chekanova

Abstract

In recent decades, e-mail has become an indispensable part of our lives. Not only is it an effortless and powerful method of communication, but it is also a highly reliable solution.

The aim of this thesis is to develop an e-mail application that allows services in the client's infrastructure to send messages to end-users.

The work contains several parts that describe the development process, which begins with gaining theoretical knowledge about e-mail and culminates in application deployment. The theoretical part covers the main principles of e-mail building, thus providing a foundation for further analysis. The second part investigates the AS-IS situation and specifies the TO-BE state; the determination of these states plays a fundamental role in defining software requirements and acceptance criteria. The development chapter highlights implemented solutions and is followed by a part that describes testing and continuous deployment approaches.

The developed application is created according to the client's requisitions and successfully deployed to the testing environment.

Keywords: Java, Spring, Spring Boot, web application, electronic mail, e-mail, MongoDB, Docker

Supervisor: Ing. Jan Zídek

Abstrakt

V poslední době se e-mail stal nepostradatelnou součástí našeho života. Nejen, že je to snadný a výkonný způsob komunikace, ale je to také vysoce spolehlivé řešení.

Cílem práce je vyvinout e-mailovou aplikaci, která umožní službám zákazníka posílat zprávy koncovým uživatelům.

Práce obsahuje několik částí, které popisují proces vývoje, který začíná získáním teoretických znalostí o e-mailu a končí nasazením aplikace. Teoretická část obsahuje principy vytváření e-mailů a poskytuje tak základ pro další analýzu. Druhá část zkoumá situaci AS-IS a specifikuje stav TO-BE, který hraje zásadní roli při definování akceptačních kritérií a požadavků na aplikaci. Dále následuje kapitola, popisující rysy nabízeného řešení a postup při jeho implementaci. Závěrečná kapitola obsahuje informace ohledně testování aplikace a zajištění kontinuálního nasazení.

Aplikace je implementována dle požadavků zákazníka a úspěšně nasazena do testovacího prostředí.

Klíčová slova: Java, Spring, Spring Boot, webová aplikace, elektronická pošta, e-mail, MongoDB, Docker

Překlad názvu: Webová služba na zpracování e-mailů

Contents

Figures

Tables

Chapter 1

Introduction

E-mail is a crucial element of social infrastructure, as it provides instant and effective communication. It is an integral part of daily life, and its use is not limited to personal areas; e-shops, banks, and other services widely use e-mails to communicate with their customers.

The application is developed for a specific customer, the Center for Knowledge Management, which in the following chapters is mentioned as CZM.

The chapter provides motivation for communication via e-mail and describes the current CZM's strategy of delivering a message to a client. The chapter ends with a definition of the thesis goals.

1.1 Motivation

Although instant messaging applications are widely used nowadays, e-mail is still an effective tool that allows fast, convenient and accurate communication. One of the benefits of electronic mail is platform independence, as the users do not need to be on the same system to exchange messages. The e-mail concept also facilitates storage and searching for information that is not currently in use: things like order details, invoices, and payment confirmation. Furthermore, electronic mail provides significant flexibility in message format, as it can support a wide range of media types.

For these reasons, an e-mail application is considered a flexible and scalable solution for establishing communication between services and end-users.

1.2 Goals

Currently, each service in the customer's infrastructure communicates with users in its own way, which leads to broad discrepancies in e-mail formats, complicated authorization processes, and code duplication. Moreover, this solution is hardly maintainable.

The thesis aims to develop a web application that simplifies message sending for the CZM's services by providing a centralized solution.

Part I

Theoretical part

Chapter 2

E-mail message format

The chapter provides an overview of e-mail format standards and message content. In addition, the reader will gain an understanding of the format of email with an attachment.

2.1 Internet Message Format

An Internet Message Format (IMF) is a standardized ASCII-based syntax for messages sent between computer users. The format requires messages to use only US-ASCII characters divided into lines. An e-mail message contains a header that is followed by a body. [1]

2.1.1 E-mail header

The e-mail header presents the information required for delivering the e-mail to the destination. The header consists of fields; each has a name followed by a colon and a body. The order of the fields is not set accurately, as they can be rearranged during transportation [1].

Table 2.1 presents the mandatory header fields, while the optional fields can be found in Table 2.2.

Field name	**Content**
From:	An e-mail address and optionally a name of the sender.
Date:	The local time and date when the e-mail was sent (set up by a sender's machine).

Table 2.1: Mandatory header fields, source: [1]

Field name	Content
To:	A recipient's e-mail address and optionally the recipient's name.
Cc:	Addresses of additional recipients, whose names are visible to each other.
Bcc:	Message's copy recipients, whose names are invisible to each other.
Subject:	A brief overview of the e-mail topic.
Message-ID:	An unique identifier for a particular version of the particular message. The host guarantees the uniqueness of ID.
MIME-version:	A version number that declares that a message supports MIME (more information can be found in Section 2.2).

Table 2.2: Optional header fields, source: [1]

Formerly, there were x-fields, where the 'X' letter stands for eXperimental or eXtension. These fields are also called non-standardized and were used for application-specific purposes. Nowadays, these fields are deprecated as the distinction between standardized and non-standardized header fields was not well defined. [1, 2]

2.1.2 E-mail body

The body of a standard text message contains lines of US-ASCII characters. There are only two restrictions: the number of characters in a line is limited to 998, and the control characters CR and LF must occur in a sequence. An e-mail with attachments is affected by more limitations, which will be discussed in Section 2.2.2. [1]

2.2 Multipurpose Internet Mail Extensions

Multipurpose Internet Mail Extensions (MIME) is an Internet standard extension of the e-mail's format that supports characters different from US-ASCII, multipart message bodies, and non-textual messages. [3]

2.2.1 Content-Type field

The Content-Type field describes the format of the body data. It allows the receiving Message User Agent (which is described in 3.3.1) to choose an

appropriate mechanism for data processing. The field contains a media type and a subtype in the following format: 'type:subtype', where both fields are mandatory. The top-level media type specifies the general type of data; the subtype specifies the format, e.g. image:jpeg. [3]

The most common types of top-level media are text, image, audio, video, and multipart. The multipart subtype can have one of the following values:

- The **mixed** subtype is used with independent body parts that should be coupled in a specific order. [4]
- The **digest** subtype is used to send collections of messages. [4]
- The **alternative** subtype is used when the system should choose the most appropriate type based on the local environment and the order of the body parts matters. [4]
- The **related** subtype is used to indicate that each part of the message is a component of the aggregation. This subtype is useful for sending a complete web page with images in a single message. [5]

2.2.2 Content-Disposition field

The MIME part can be displayed inline with the Content-Disposition set to "inline" value as the following:

```
Content-Disposition: inline;
```

The MIME part can be set to display the attachment automatically with:

```
Content-Disposition: attachment;
```

Nevertheless, most of the e-mail agents do not follow these instructions and use internal algorithms to determine the multi-purpose Internet Mail Extensions parts. [6]

2.3 E-mail with an attachment

An e-mail, which includes an attachment, is made up of multiple parts. A multi-part e-mail contains one or more different data sets combined in a single body. The entity's header must contain a 'multipart' Content-Type; the body contains one or more parts separated from each other by an encapsulation boundary. If there are no headers in the body parts, the blank line will follow the boundary string. The encapsulation boundary should be explicitly specified in the content-type field. [4]

An example of a specification of the encapsulation boundary:

```
Content-type: multipart/mixed; boundary="my boundary"
```

A multipart message can look as follows:

```
From: Someone <someone@example.com>
Date: Thu, 9 Dec 2021 13:24:57 +0100
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary="my boundary"

The mail composers include there
an explanatory note for readers incompatible with MIME.
This text will be ignored.

--my boundary
Content-Type: text/plain

It is the text body.

--my boundary
Content-Type: text/plain;
Content-Disposition: attachment;
        filename="example.txt"

It is the message attachment.
--my boundary
```

Chapter 3

The Principles of E-mail Processing

The chapter describes the mechanisms of e-mail processing and contains DNS-related terms. The reader will become acquainted with e-mail related protocols and e-mail agents.

3.1 Terminology

A **Mail Exchanger Resource Record (MX RR)** is a resource DNS record used for e-mail transfer based on the Simple Mail Transfer Protocol. More information can be found in 3.2.3. The record contains two fields: [7]

- The **host name** points to an address DNS record.
- The **priority** defines the record that should be used. The preferred record is the one with the lowest value.

An example of MX RR is presented in Table 3.2:

Domain	TTL	Class	Type	Priority	Host
example.com	3600	IN	MX	5	mail.example.com

Table 3.1: Example of MX RR

A **Fully qualified domain name (FQDN)** is a domain name that specifies its exact location in the DNS's tree hierarchy. It contains all domain levels, including the top-level and root zones. FQDN can be interpreted exactly one way. For a device with the hostname 'host' located in the parent domain 'parent.com', the fully qualified domain name is 'host.parent.com'. [8]

A **Mailbox** is a place for e-mail storage. It is a conceptual entity, which is defined by an e-mail address in the following format: username@domain. [1]

3.2 E-mail protocol

An e-mail protocol is an established set of rules for e-mail transmission. There are three main types of e-mail protocols; all of them belong to the application layer. The message transferring is based on the TCP connection. Therefore,

all of these protocols are reliable, which means that the data will be delivered in the correct order. These e-mail protocols are not secure by default, so the use of cryptographic protocols is desirable. [9, 10]

3.2.1 Post Office Protocol

The Post Office Protocol (POP3) is the most common version that uses port 110 for unsecured communication. The POP server is installed on the recipient's mail server, while the POP client is installed on the recipient's machine. The protocol provides support for the 'offline' paradigm. Once the e-mails are received from the mail server on the client's device, there is no need for an Internet connection. [11]

The process of client authorization starts after receiving the following command from the server:

```
OK POP3 server ready
```

The client sends a username followed by a password, then the mail server validates the credentials, and sends an OK message in case of success. When the connection is established, the client has access to the list of mails located on the mail server. E-mails are deleted from the server after they have been retrieved from the server to the client's machine. Thus, accessing the same e-mails on another device cannot be achieved without an explicit configuration of the copying strategy. [12]

3.2.2 Internet Message Access Protocol

Internet Message Access Protocol (IMAP) is an Internet standard protocol used by e-mail clients to retrieve e-mails from a mail server. Unlike the Post Office Protocol, IMAP provides full remote access to a mailbox, in addition to content retrieval. Unsecure communication is configured on port 143. [13]

Features (in comparison to the POP3) [14] :

- IMAP enables parallel access to the mailbox for multiple users. It is achieved by keeping the message on the mail server even when the user has already downloaded it.
- IMAP enables tracking the message's state by setting flags such as "seen", "answered", "deleted", and "draft". Flags are stored on the mail server, so different users accessing the same Mailbox see consistent information.
- IMAP client stays connected to the mail server as long as the user interface is active; POP client connects to the mail server, receives e-mails, and disconnects.
- IMAP enables downloading of only selected MIME parts. Therefore, the client can download the message text without downloading the attachment.

- IMAP enables one to perform server-side searches. Therefore, the client does not need to download messages from its own Mailbox to find information.
- IMAP enables mailbox manipulations such as creating, renaming, and deleting on the server.

3.2.3 Simple Mail Transfer Protocol

Simple Mail Transfer Protocol (SMTP) is a text-based internet communication protocol used to send and receive messages. The protocol provides connection-oriented communication, which means that the communication session should be created prior to data transfer. The protocol uses port 25 for unsecured communication. [15]

In contrast to POP and IMAP, which are designed to receive or 'pull' e-mail from the mail service, the Simple Mail Transfer Protocol is mainly used to send or 'push' e-mail from one mail server to another. The routing of e-mails is based on the achievement of the destination server, not the individual user. Message transmission through SMTP occurs within the session originated by an SMTP client. The initiating host is either an e-mail client or an SMTP server (mail relay) that acts as an SMTP client. [15]

Formerly, SMTP servers provided access for a user whose IP address was in the mail server's field of knowledge. Today, the vast majority of servers require authentication with credentials before granting access. Security issues related to e-mail transmission will be described in section 4.4. [16]

3.3 E-mail Agent

An e-mail agent is part of the e-mail processing infrastructure, which includes creating the sender, transferring over the network, and viewing by the recipient. [17]

3.3.1 Message User Agent

A Message User Agent (MUA, email client) is software on the client's computer to write, send, and view e-mails. The program transforms the text written by the sender into the appropriate e-mail message format. Examples of e-mail clients: Gmail, Thunderbird, Evolution, Spark, etc. [18]

3.3.2 Message Submission Agent

A Message Submission Agent (MSA) is software that accepts messages from the e-mail client. It either delivers them or acts as an SMTP client to relay them to a Message Transfer Agent (which is described in 3.3.3). Today, MSA does the final preparation, including adding header fields such as Date and Message-ID. The agent validates the address in the 'Sender' field and checks

the message's format. In case of failure, it immediately enforces the sender to fix the error. [19]

3.3.3 Message Transfer Agent

A Message Transfer Agent (MTA, SMTP relay, mail server or Mailer) is software that is responsible for choosing the appropriate route for e-mail based on the Mail Exchanger records. It can act either as a Boundary MTA (relays the message to other MTAs) or as a Final MTA (transfers the message to the MDA). [17]

A mailer at the local domain prepares a query for MX RRs for the destination domain. The response is a list of MX RRs for the destination domain. The Mailer removes irrelevant RRs from the list and attempts to deliver the message to the MX with the lowest preference value. MTA keeps trying until the MX accepts the message or there are no untested MXs. Examples of MTA: Postfix, Exim, Sendmail, or qmail. [20]

3.3.4 Message Delivery Agent

A message delivery agent (MDA, local delivery agent) is software that delivers the message from MTA to the Mailbox. The MDA's function is to determine the internal location of the Mailbox for performing the delivery. [20]

3.4 Example of e-mail processing

An example of events occurs when abstract sender Pierre sends a message to abstract recipient Andre using a mail client: [7, 15]

1. The MUA formats the IMF from Pierre's message and sends the message to the local Mail Submission Agent using SMTP.
2. The MSA accepts the e-mail submission and checks for errors. The agent then performs preprocessing and relays the message to the Message Transfer Agent for further transmission.
3. The MTA relays the message to other MTAs. Once the sending sequence is completed, the message reaches the Message Delivery Agent.
4. The MDA stores the received e-mail in Andre's Mailbox.
5. Andre's MUA receives the e-mail from the Mailbox using the IMAP/POP protocol.

The described process is presented in Figure 3.1:

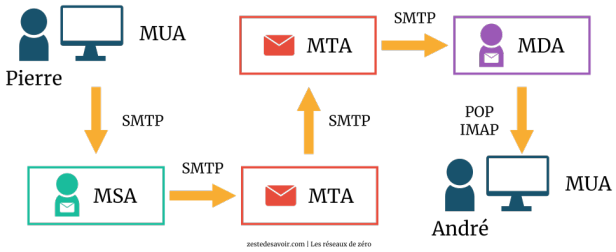

Figure 3.1: E-mail processing diagram, source: [21]

Chapter 4

E-mail security

The chapter describes the authentication methods that are used prior to email submission in order to verify a user's identity. In addition, this part highlights vulnerabilities of email transmission.

4.1 Simple Authentication and Security Layer

A Simple Authentication and Security Layer (SASL) is a framework that provides authentication support for the application layer protocols. It presents an abstraction layer between protocols and authorization mechanisms. This allows application protocols to use SASL without redesigning the protocol. Often, application protocols should support TLS for using SASL authorization services. Usually, only one successful authentication exchange may occur in a protocol session (unless an explicit definition is included in the protocol's technical specifications). After successful authentication exchange completion, further authentication attempts fail. [22]

SASL provides a wide range of authentication methods. The connection should be SSL-encrypted for using the PLAIN or LOGIN method. [23]

SASL mechanisms:

- PLAIN. The username and password are transmitted as one string encoded in Base64. [24]
- LOGIN. The username and password are encoded in Base64 and transmitted separately. [24]
- CRAM-MD5. It is a challenge-response authentication mechanism. The password is not transferred to the server. Instead, the server provides the client with a computational task that can only be solved with the help of a password. The task is different for every login, so spammers cannot misuse data from previous connection to the server. [25]
- XOAUTH2. The solution is based on OAuth signatures. It is widely used in systems such as Gmail and Outlook.com. [26]

4.2 Extended Simple Mail Transfer Protocol

An Extended Simple Mail Transfer Protocol (ESMTP) is an extension of SMTP. It provides support for the new commands: [27]

- EHLO initiates ESMTP. It is an alternative to SMTP's HELO command.
- STARTTLS enables sending multiple commands in a batch without waiting for a response to each one.[28]
- AUTH enables authentication mechanism.
- SIZE declares the message size. Thus, the server may indicate to the client whether it is willing to accept the message.

4.3 SMTP authentication methods

For verifying that the user has a right to send an e-mail, the mailer must perform an authentication. There are several SMTP authentication methods, but authenticated SMTP is the most widely used today. [27]

4.3.1 IP address restrictions

Authentication by IP address is the simplest method based on gaining access to preset IP addresses. They can be easily spoofed unless all transport paths between the mail client and Message Transfer Agent are trustworthy. The use of this method is especially problematic for dynamic IP addresses. [16]

4.3.2 Prior POP authentication

Sometimes this method is called 'POP before SMTP'. To perform this type of SMTP authentication, a user must connect to the POP3 service of the same e-mail server and authenticate. Then the client immediately connects through SMTP to the same server. The SMTP server remembers the user's IP address and provides authentication. An advantage of the method is that POP and SMTP are widely adopted. Therefore, there is no need to update the client's software. Implementation is not difficult, and the method provides great flexibility. [16]

The method problems defined, which are defined in the article [29], can be found below:

- Someone uses IMAP instead of POP.

- After disconnection from the Internet or changing IP address by the ISP, the hacker can allocate the IP address and send e-mail from the account.
- More than one client can share the same IP address using NATs and proxies.
- Both the POP3 and the SMTP services should run on the same system.
- MSA should communicate with the POP server, making the implementation more difficult.

However, the solution persists on old servers.

4.3.3 Authenticated SMTP

Authenticated SMTP (SMTP AUTH) is an SMTP service extension that provides an authentication mechanism. It allows the Message Submission Agent to validate the authority and determine the identity of the submitting user and must be supported by the MSA. The implementation uses the security mechanisms provided by SASL. Instead of the classical SMTP port 25, the authenticated SMTP uses port 587. ESMTP authentication adds session orientation to the SMTP. [27]

An example of SASL-based authentication: [30]

```
250 AUTH LOGIN PLAIN CRAM-MD5 #server responses
#with available types of auth methods
AUTH LOGIN #client chooses the authentication type
334 dXNlcm5hbWU= #server requests a username
dmFjbGF2 #client provides the username
334 cGFzc3dvcmQ= #server request a password
c2VjcmV0 #client provides the password
235 Authentication successful #server validates credentials
```

4.4 Secure message transmission

Even though user's credentials can be encrypted using SASL mechanisms or alternatives, there is still a need to encrypt message transport. In the case of unsecured message transmission, there is a chance of a man-in-the-middle attack when the attacker can read and modify the communication. [31, 32]

The concept of using SSL/TLS with e-mail protocols is similar to HTTPS, as it wraps HTTP inside TLS. This allows the mail client and Mail Submission Agent to protect message submission, integrity, and privacy. A naming convention, which adds "S" at the end of the protocol's name (e.g. SMTPS, IMAPS), indicates that the protocol is based on TLS encryption. TLS can be enabled in two different ways: [33]

- Implicit TLS is negotiated immediately once the connection has been started on a separate port. This method is used more widely for IMAP and POP. IMAP port for implicit TLS is 993, the POP port is 995, and the SMTP port is 465. [34]
- Explicit TLS. This method enables TLS by using the STARTTLS command, which upgrades a plain text connection to the encrypted connection and does not require the use of a separate port. If the client issues a STARTTLS command, a TLS handshake upgrades the connection. Unlike the IMAP and POP3 protocols, explicit TLS is more common for SMTP. [35]

Part II

Analysis

Chapter 5

Analysis

This chapter describes drawbacks of the pilot application and defines the requirements and acceptance criteria for the new e-mail service.

5.1 The pilot version

The pilot implementation aimed to provide a centralized solution for e-mail delivery inside the CZM's ecosystem. The functionality of that version is limited to sending an e-mail received in JSON format on the endpoint. The application contains a REST layer auto-generated by the OpenAPI generator and the service layer responsible for e-mail processing. Due to the limited functionality of the pilot version, e-mail building is the only method remaining in the new application.

5.2 Application usage

In order to deliver a solution that meets its requirements, we need to understand how the application will be used. The CZM's analytics provided me with the following details: "Each application inside the ecosystem should be able to authenticate into the e-mail service and, according to its permission, send an e-mail in a desirable way." Based on the received information, I specified the requirements provided in the following section.

5.3 Application requirements

This section contains the functional and non-functional requirements for the e-mail service.

5.3.1 Functional requirements

FR1. Authentication mechanism.

FR2. Roles distinction: an administrator and a standard user.

FR3. Sending e-mails:

FR3.1. Sending e-mails without a configuration.

FR3.2. Sending e-mails with a specified configuration.

FR4. E-mail storage and browsing.

FR5. E-mail configuration support:

FR5.1. Configuration can be added, removed, and updated in the system.

FR5.2. Access to a particular configuration can be granted to a user.

FR5.3. A user has his own list of available configurations.

FR6. Support for using different mailers.

FR7. Sending e-mails with attachments.

FR8. User management:

FR8.1. A user can be added or removed from the system.

FR8.2. User's credentials can be changed.

FR8.3. An administrator can change the user's role in the system.

5.3.2 Non-functional requirements

NFR1. The application is scalable.

NFR2. The application can be used by several users simultaneously.

NFR3. The application contains sufficient test coverage.

NFR4. The application provides access to the logs received from the test environment.

NFR5. The application follows Continuous Delivery approach.

NFR6. The application has low latency and response time.

NFR7. The application provides Swagger documentation.

5.4 Acceptance criteria

In order to answer the question "Does the application meet the requirements?" we need to ensure that the product is suitable for its intended purpose and follows the contracted obligations. The setting of acceptance criteria is a widely used practice to evaluate completed work in the final phase of development. Therefore, the following acceptance criteria were defined:

- **User**:
  1. A user who provided valid credentials is authenticated in the system.

2. An administrator adds and removes the user from the system.
3. A non-administrator cannot add and remove user from the system.

- **E-mail**:
  1. A user sends an e-mail without specifying configuration.
  2. A user sends an e-mail using one of their permitted configurations.
  3. E-mails which were sent by the application can be viewed by the administrator.
  4. An e-mail can contain an attachment.
- **Configuration**:
  1. An administrator can add and remove configuration from the system.
  2. An administrator grants access to the configuration for a user.
- **Mailer**:
  1. The application supports using different mailers.

5.5 Existing solutions

There are a couple of existing solutions for e-mail sending services, such as Postmark, Amazon Simple Email Service, or Mailjet. They offer free plans with limitations (e.g., only the first 6 000 e-mails per month are free). There is also a limit on storage around 500MB, which is not enough for our purposes, as e-mails may contain pretty huge attachments. Due to the CTU information [36], there are around 18 000 students in the university, and even under the assumption the service does not send e-mails to all students simultaneously, it can be clearly seen, 6 000 free e-mail is above free limit. Moreover, the external solutions cannot be modified in case of a need for additional functionality. To conclude, existing solutions do not satisfy the customer's requirements, and there is a need to develop a custom solution.

Chapter 6

Application design

The application architecture is one of the essential things software engineers should arrange before starting the development process. It helps facilitate service maintenance and improves its extensibility. Furthermore, the earlier the application design is produced, the lower the service development costs.

The chapter provides an overview of the major architectural solutions, used technologies, and ways of data treatment.

6.1 Architecture

This section covers the main architectural solutions, such as modules and layers separation.

6.1.1 Layers

The web application can follow a single- or multi-layer architecture. Although the one-tier approach is easier to implement, it has a significant drawback, namely the complexity of its maintenance. Therefore, I decided to separate the layers to facilitate high scalability. Thus, the application has traditional for a web application layers:

- The **REST** layer contains controllers that will be exposed to a client. Requests are not processed here; instead, they are passed over to the next layer.
- The **service** layer contains business logic.
- The **data** layer is responsible for the communication with the database.

Based on modern practice, the underlayer cannot directly access the top layer. For example, the data layer cannot call the REST layer. Figure 6.1 shows the described architecture:

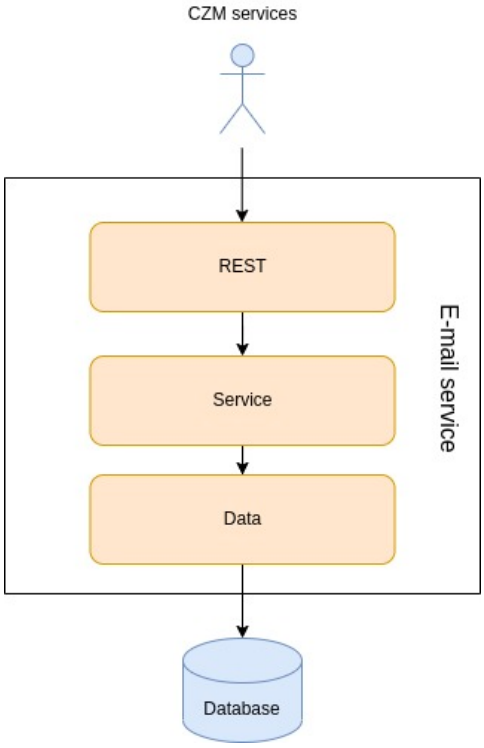

Figure 6.1: Application layers

6.1.2 Modules

Decomposing a monolithic Java application into modules helps to achieve strong encapsulation and improve scalability. Therefore, I defined the modules that are presented in Table 6.1:

Module	Purpose
App	Holding the application configuration (spring, docker, logs)
Data	Ensuring storing in a database
Service	Building and sending e-mails
Rest	Holding REST endpoints definitions

Table 6.1: Application modules

6.2 Choice of technologies

The section contains a comprehensive overview of the development, testing, and deployment technologies used in the e-mail service. The choice of these tools is made on the basis of existing standards and preferences in CZM.

- Java 11 is an object-oriented, platform-independent language that has gained a lot of popularity in recent years. [37]
- Spring Boot Framework provides great support for data management and deals with security issues. In addition, Spring simplifies logging and configuration.
- GitLab platform tracks code modifications and handles Continuous Delivery.
- Docker Swarm enables container orchestration.
- JUnit 5 and Mockito frameworks allow writing automatized tests. Moreover, Postman is used to testing REST API.
- SonarQube static code analyzer helps to meet code standards by evaluating code against the quality rules; moreover, this tool can detect security vulnerabilities.
- Swagger Documentation allows visualization of API's resources.

6.3 Domain diagram

The following entities were identified based on the functional requirements:

- **User** represents the service that has access to the e-mail application.
- **E-mail** presents the message sent by the application.
- **Configuration** contains the details of how e-mail should be processed, e.g. whether it should be logged, sent by a real mail server, etc.
- **Mailer** holds credentials for SMTP authentication.
- **Attachment** presents e-mail attachment. It contains an array of bytes that presents content and the MIME type that allows MUA to process the file.

The domain diagram of the e-mail service is presented in Figure 6.2.

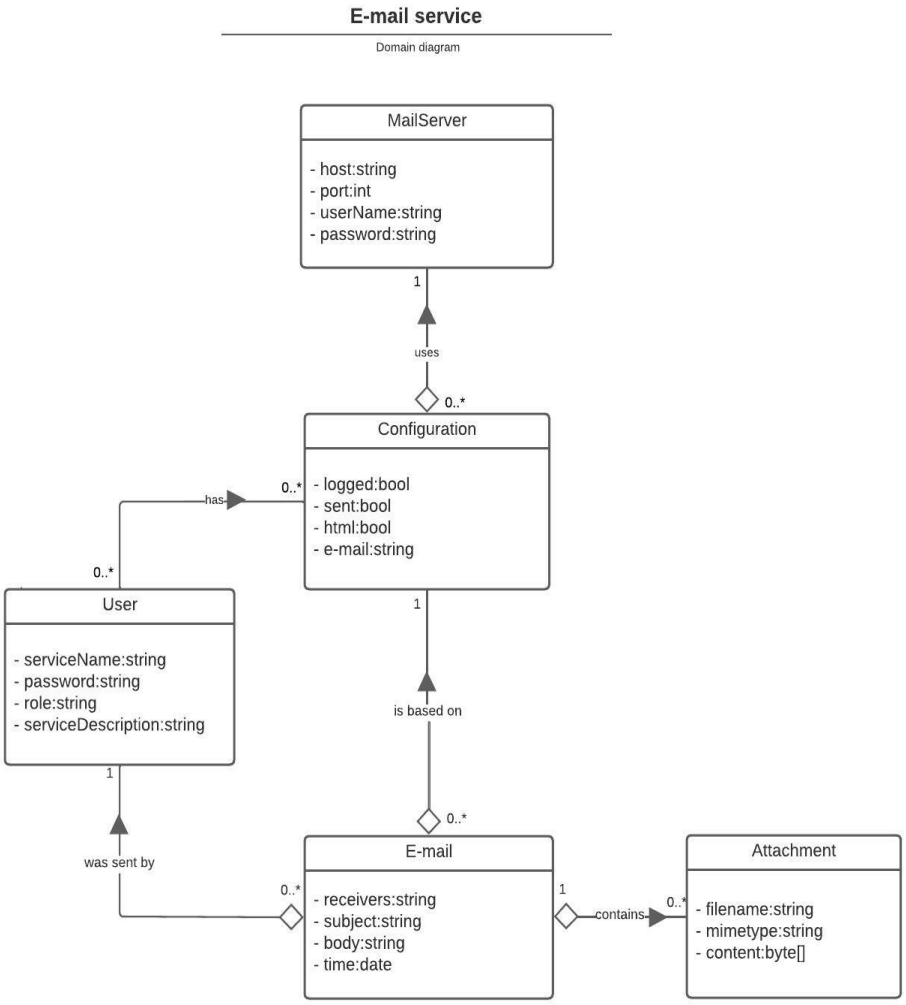

Figure 6.2: Domain diagram

The developed design ensures low coupling and enables one to add new features without complex design changes.

6.4 Data

The application should enable e-mail storage and use e-mail configurations. Furthermore, the e-mail service should provide authentication and authorization mechanisms, which also require database usage.

Therefore, the pilot version neet to be extended by a data module responsible for database operations and object representation.

6.4.1 Database type

E-mails can have abnormal forms, since they may contain an attachment, and the receiver field might hold multiple entries. An e-mail can be built in different ways; moreover, it may present a wide variety of information. Therefore, the use of a predefined database table structure is not appropriate for our case, and I decided to use the NoSQL database.

One of the advantages of the NoSQL database is that it does not use normalization, which speeds up queries, as all required data is stored together. Moreover, this database concept avoids joins and can be easily scaled horizontally. Another advantage is data distribution, which means that there is no single control unit. This leads to continuous data availability. [38]

Data duplication is a possible drawback, but due to the expected size of application usage, this will not impact the service performance. The other possible disadvantage is the lack of consistency when performing multiple transactions simultaneously. However, database providers often offer their own concurrency control mechanisms to ensure consistency. [38]

6.4.2 NoSQL storage type

There are four main types of NoSQL databases:

- Key-value. This type is considered the least complex, as it resembles a relational database with only two columns: an attribute name and a value. This type of storage is widely used for shopping carts and the implementation of user preferences. Some other examples include Redis and Dynamo. [39]

- Column-based. Each column is treated separately, whereas single column values are kept adjoining. The benefit of this data storage is fast reading, as columns often have the same type, whereas the possible drawback is complicated writing operations. Therefore, the solution is prevalent on analytic platforms. Among several examples, the most popular ones are Cassandra and Google's Bigtable. [40]

- Graph-based. The central focus is put on the relationships between entities. The scheme is presented by connected nodes. Each node contains a direct link to the adjacent elements, making index lookups unnecessary. This type of database offers optimization for traversing connected data. Therefore, its niche includes social networking websites as well as recommendation engines. Some of the commonly used graph databases are Neo4j and Nebula Graph. [39]

- Document-based. This type of database stores data in JSON, BSON, or XML documents and supports document nesting and indexing elements.

> The document-based database reduces the amount of transactions by providing a more natural way of storing data objects. This approach is widely used in trading and blogging platforms. [40]

Since the e-mail service tends to work with abnormal data types and does not need to concentrate on the relations between the entities, I chose document-based storage as the most appropriate.

6.4.3 Document-based database

There are many document-based databases. The most popular are Google Cloud Firestore, Amazon DynamoDB, Microsoft Azure Cosmos DB, Couchbase, and MongoDB. [41] Nevertheless, the Spring framework provides support only for Couchbase and MongoDB. Compared to Couchbase, ACID transactions in MongoDB can be performed more easily. Although the Couchbase query language is similar to SQL, MongoDB provides incomparable versatility in data management. Therefore, MongoDB is a more suitable solution for the e-mail service.

6.4.4 Database design

Even though MongoDB encourages the preference of embedded documents over references, this approach is used primarily with aggregation relations and is not always the most appropriate. Based on the analysis, I chose the following strategy: all relations between entities will be held using references. The exception is the relation E-mail - Attachment, where we can save Attachment as an embedded document within an E-mail.

Part III

Development

Chapter 7

Data and service layers

Building data layer is one of the milestones in application development, although this implementation does not produce immediate visible results. The service tier contains business logic that defines the application behavior; moreover, it plays a crucial role in enabling requests received on the REST tier to perform actions in the data layer.

This chapter provides the reader with the central concerns regarding data storage and implementation of business logic in the service layer.

7.1 Data layer

One of the main purposes of the data layer is to ensure communication with the database. This section clarifies the choice of approaches taken in data management. The information includes details of the validation of entities and holding relations in a database.

7.1.1 Identifiers

MongoDB enables the storage of IDs in two different ways. The first one is Mongo's BSON implementation, ObjectId. Another option of keeping ID is the use of a plain Java String.

For our purposes, I consider using the built-in ObjectId to be a better option, as it only takes 12 bytes while the hex represents requires 24 bytes. In addition, this Mongo solution provides a powerful indexing mechanism that speeds up inserts, as only the latest index part is loaded to ensure uniqueness. Moreover, ObjectId contains a timestamp, which is convenient for accessing the date of sent e-mails. [42]

As stated in the documentation, MongoDB will automatically create an ID of ObjectId type if a developer does not specify it. [42] Nevertheless, I explicitly declared ID to avoid ambiguities. To avoid adding an ID attribute to each entity, which causes code duplication, I created an abstract class containing the ID attribute, so that the successors of this class do not need to declare an identifier once more.

The code of `AbstractEntity` is presented below:

```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@SuperBuilder
public abstract class AbstractEntity {
  @Builder.Default @Id private ObjectId id = new ObjectId();
}
```

7.1.2 Reference types

Mongo provides two reference mechanisms:

- **Manual references** require keeping the object's `_id` in another document as a reference. Relevant data can be fetched by running an additional query. [43]
- **DBRefs** in addition to the ID, keep the collection name, and, optionally, the database name. Moreover, Spring Data provides great support for this reference type by the annotation `@DBRef`. Thus, it eliminates the need to run queries manually, which makes the usage of a simple getter method sufficient. [44]

Furthermore, Spring offers one extra method for referencing, which works slightly differently, even though it resembles DBRefs:

- `@DocumentReference`. The main concept is the same as in `@DBRef`. The difference is that anything can be used as a reference, from a single value to an entire document. [44] Nevertheless, this feature was only introduced on Spring Data MongoDB 3.3.0, while CZM uses the older version of the library. Therefore, to avoid modifications to other applications, *@DocumentReference* is not an appropriate solution.

In conclusion, `@DBRef` is considered the most effective solution, as it provides great flexibility and does not require a change in the CZM infrastructure. Therefore, the one-to-many relation is declared as follows:

```
...
public class User extends AbstractEntity {
    ...
    @DBRef private List<Mail> sentEmails;
}
```

The sender name in the Mail entity is a natural ID, as it is unique. Therefore, the field `sender` in `Mail` provides a back reference to the `User` entity.

The drawback of referencing in MongoDB is the lack of cascading mechanisms, which causes in requirement to handle parent-child relations manually:

```
List<Mail> mails = user.getSentEmails();
mails.add(mail);
user.setSentEmails(mails);
```

7.1.3 Validation

MongoDB indices can be used to ensure the uniqueness of fields at the data layer. This feature is also present in the Spring Data Library via the annotation `@Indexed (unique = true)`. In addition, it supports imposing uniqueness on field combinations. This functionality is used to avoid duplicates of the mailers in the following way:

```
@CompoundIndex(
    name = "mailer_idx",
    def = "{'host' : 1, 'port' : 1, 'username' : 1, 'password' : 1}",
    unique = true)
public class MailServer extends AbstractEntity {...}
```

Moreover, to ensure field validation on the database side, I used the @NotBlank annotation from javax.validation.constraints.

7.1.4 Repository pattern

Repository pattern has gained popularity in recent years due to its advantages in ensuring loose coupling and enabling dependency injection. The mentioned features improve application testability, which is beneficial due to CZM's requirement on test coverage. Moreover, this pattern restricts direct access to the data layer from the controller, therefore providing the abstraction between the REST layer and the database context. [45]

Spring Boot contains a Data project, which provides implementation of the Repository pattern out of the box. Moreover, there is a predefined set of keywords for the method name, which are automatically processed by Spring, and the `@Query` annotation that allows one to write natural Mongo queries:

```
public interface MailRepository extends MongoRepository<Mail, Long> {
  @Query("{ 'sender' : ?0}")
  List<Mail> findUsername(String sender);
}
```

7.2 Service layer

This section contains a comprehensive overview of the design approaches implemented at the service layer. First, I provide the motivation for the Abstract service and present its implementation, and then, I discuss the aspects of e-mail building and processing.

7.2.1 Abstract service

Many services have the same implementations of the basic methods, which results in code duplication. In order to avoid it, I implemented an `AbstractService` with create, read, update, delete (or CRUD) operations. Different repositories (`UserRepository`, `ConfigurationRepository`...) are in use, since there are

several entities. The use of Java Generics, allowing abstract classes to handle different object types, facilitates the fulfillment of the requirement. Since all entities extend the `AbstractEntity` I declared an upper bound `<T extends AbstractEntity>`. In this way, any child of `AbstractEntity` can be used. The code of `AbstractService` can be found below:

```
public abstract class AbstractService<T extends AbstractEntity> {

  private static final Logger log =
  LoggerFactory.getLogger(AbstractService.class);

  public abstract MongoRepository<T, ObjectId> getRepository();

  public void create(T obj) {
    getRepository().save(obj);
    log.info("{} was added", obj);
  }

  public Optional<T> findById(ObjectId id) {
    return getRepository().findById(id);
  }

  public void delete(ObjectId id) {
    getRepository().deleteById(id);
    log.info("Entity with id={} was deleted", id);
  }

  ...
}
```

To obtain the appropriate repository instance, it is necessary to define a getter in child classes explicitly:

```
  ...
  public class ConfigurationService extends AbstractService<Configuration> {
    private final ConfigurationRepository repository;
    @Override
    public MongoRepository<Configuration, ObjectId> getRepository() {
        return repository;
    }
  ...
  }
```

To sum up, this approach improves code reusability as we can define the method once and allow other classes to use it. Furthermore, if there is a need for a change in the core method, it is sufficient to modify the code once without affecting successors. Thus, it reduces the risk of programming errors.

7.2.2 E-mail building

- **Simple Java Mail** library facilitates the creation of e-mails, as it provides support for attachments and MIME messages. In addition, it is an RFC-compliant solution. The code that uses Simple Java Mail looks clean and is easy to understand:

```
new MailBuilder()
  .setFrom(configuration.getEmail())
  .addTo(mail.getReceivers())
  .setSubject(mail.getSubject())
  .addBody(mail.getBody(), configuration.isHtml())
  .addAttachment(mail.getAttachments())
  .send(mailer, encryptPass(configuration));
```

- **Builder and Facade design patterns**. The old version of the application needed to decide whether the e-mail will be logged or sent according to the active environment. For this purpose, the pilot version used the `MailBuilderFacade` class that implements Builder and Facade design patterns. The new version of e-mail service moves this functionality to the Configuration entity. Thus, this pattern is no longer needed. Nevertheless, there is still a need for the builder pattern that can be justified by the presence of ample e-mail parameters.

7.2.3 Entity validation

Eliminating the consequences of unexpected user behavior is a crucial part in ensuring smooth running of the application. Therefore, it is necessary to implement a data validation mechanism. For example, to avoid attempts to send the e-mail using the configuration without access to a mailer, the following actions should be taken:

```
public void validateConfiguration(Configuration c) {
  if (c.isSent() &&
  (c.getMailServer() == null || c.getEmail() == null)) {
    throw new EntityConfigurationException(
        "In order to send an e-mail mail server and
           e-mail sender should be specified in the configuration");
  }
  log.info("configuration was successfully validated");
}
```

There is further validation of entities on the data side, which was described in Section 7.1.3.

Chapter 8

REST layer and security

This chapter describes the way the REST layer is implemented and provides an overview of applied security mechanisms.

8.1 REST layer

The REST layer is exposed to the client, therefore, it should be handled especially carefully. The application has to be secured from a malefactor, and the end-client potentially needs to get a detailed description of thrown exceptions.

This section covers main concerns regarding the REST layer, such as processing requests and responses using a Data Transfer Object and handling exceptions. In addition, an overview of the Swagger specification is provided.

8.1.1 Data Transfer Object

Data Transfer Object, or DTO, is designed to address the problem of excessive calls between a client and a server by aggregating the data. Moreover, this approach allows us to have different views by decoupling the model and its representation. Therefore, a user does not get the complete object; they will get only the information that they are allowed to see. Also, it is important to carefully handle client requests, as the application uses a NoSQL database with no restrictions on database schema. Only harmless fields from the REST request should reach the service layer. [46]

Although mapping from DTO to plain model and vice versa can be performed manually, I used the ModelMapper library, which automatically handles this process. The setup is simple and requires only the configured `ModelMapper` bean. Then a controller should contain a conversion method. An example is presented below:

```
private Mail convertToEntity(MailDto mailDto) {
  return mapper.map(mailDto, Mail.class);
}
```

This pattern is especially beneficial for User representation, which holds sensitive credentials such as a password that cannot be exposed to the REST

layer. In the future, this pattern will find more uses. [46]

8.1.2 API design

The path parameter identifies a specific resource; while some methods additionally require an ID. Moreover, the method can contain a body in JSON format. Therefore, the format of endpoints is the following:

- /configurations
- /users
- /configurations/6256b6156eaae53ea65cee6a

Overall, the API uses classical REST API methods such as Post, Get, Put, and Delete, and path parameters contain entity names in plural form. An example of a full request can be found below:

```
POST http://{baseurl}/configurations
Authorization: Basic camunda service
Content-Type: application/json
{
    "name": "new-",
    "email": "test@email.com",
    "isSent": true,
    "useHTML": true
}
```

8.1.3 Exception handling

Spring framework provides some enhancements to Java handling exceptions methods. These mechanisms can be specified globally at the application level or locally for each class. To facilitate maintenance, I decided to implement global exception handling. In this way, there is a main class that is responsible for exception handling. If there is a need for a change, it suffices to change the code in one place. Thus, this approach reduces the code boilerplate.

All undefined exceptions are handled by the default method, which returns custom `ErrorResponse`:

```
@Getter
@Setter
public class ErrorResponse {
  private final String message;
  private Instant timestamp;

  public ErrorResponse(String message) {
    this.message = message;
    this.timestamp = Instant.now();
  }
}
```

There is a need to specify the exceptions in `GlobalExceptionHandler`, so it might look as follows:

```
@ControllerAdvice
@Slf4j
public class GlobalExceptionHandler extends ResponseEntityExceptionHandler {
  ...
  @ExceptionHandler(NoSuchElementException.class)
  public ResponseEntity<ErrorResponse>
  handleNoSuchElementFoundException(NoSuchElementException ex) {
    log.error("Failed to find the requested element", ex);
    return buildErrorResponse(ex, HttpStatus.NOT_FOUND);
  }
}
```

8.1.4 Swagger

The pilot version followed the API-first approach. There was a `spec.yaml` file that contained API documentation and instructions to generate controllers.

The drawback of that method was the inconvenience of extracting basic-auth credentials from the header. It occurred because the controllers were auto-generated and therefore could not be changed. For that reason, it was impossible to pass a header from the controller to the service layer. Therefore, the API-first approach was considered unflexible for our purposes, and it was superseded by the Build-first one, where the documentation is generated based on the existing controllers.

The existence of API documentation is granted by the Swagger.v3 annotations. Among them in the Rest layer are `@Operation`, `@ApiResponse`, `@Schema`; on the model layer, the annotation `@NotNull` was used to provide a view of the required fields.

The generated documentation provides a great base for application extension. Swagger output can be used to generate the API client to make the application available to real services.

8.2 Security

Security is one of the most critical issues that should be addressed in the early stages of development. This section clarifies the decision about implemented security mechanisms.

8.2.1 Authentication

Authentication is one of the functional requirements, and Spring framework has excellent support for it.

The pilot version used Bearer token authentication. A token of each request was compared with the one defined in `application.properties`.

This solution would not suit the production version, as there is a requirement to have several users. Moreover, this approach does not provide much security.

The Spring security library contains many authentication methods, starting with LDAP and ending with SSO. For e-mail service purposes, Basic authentication covers all security requirements and therefore is considered sufficient.

To ensure that requests are authenticated, I implemented `org.springframework.security.core.userdetails.UserDetails;` with the BCrypt hashing mechanism. Then I prepared the following configuration:

```
...
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter
implements WebMvcConfigurer {

  ...
  private final UserService userDetails;

  @Autowired
  public void configureGlobal(AuthenticationManagerBuilder auth)
  throws java.lang.Exception {
    auth.userDetailsService(userDetails).passwordEncoder(passwordEncoder());
  }

  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .anyRequest()
        .authenticated()
        .and()
        .httpBasic()
        .authenticationEntryPoint(entryPoint)
        .and()
        .csrf()
        .disable();
  }

  @Bean
  public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
  }
}
```

8.2.2 Authorization

According to the defined requirements, the e-mail service should support the user and administrator roles. The authorization mechanism is implemented

using a built-in Spring Boot mechanism. This allows limiting access to methods by using `@PreAuthorize("hasAuthority('ADMIN')")` annotation on the rest methods. In order to use it, we should ensure that the user's authorities are retrieved properly. It was achieved by rewriting `getAuthorities` as follows:

```
@RequiredArgsConstructor
public class UserPrincipal implements UserDetails {
...
    private final User user;
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        List<GrantedAuthority> list = new ArrayList<>();
        list.add(new SimpleGrantedAuthority(this.user.getRole()));
        return list;
    }
}
```

8.2.3 Encryption

One of the functional requirements is sending e-mails using the specified mail server. To achieve this, the mailer's connection credential must be stored in the database. Although keeping passwords as plain text is a widely used practice, the e-mail service stores passwords encrypted. Encryption is provided by symmetric AES cipher, more specifically, the `ECB/PKCS5Padding` scheme. Although using a stronger scheme, such as CTR or GCM is preferred, ECB is sufficient for the current stage. The key is saved in the application properties but will be moved to the external Systems Password Manager in the future.

Part IV

Final phases

Chapter 9

Testing

The purpose of testing is to verify expected behavior, improve code quality, and ensure that new changes will not break existing functionality. This chapter discusses the methods used to test the application. Additionally, the meeting of the defined requirements and acceptance criteria is verified.

9.1 Test approach

This section explains the choice of testing techniques by describing its benefits and drawbacks.

9.1.1 Test execution

The choice of test execution method depends on several factors, among them, project requirements and the team's expertise. The section compares the manual and automated testing approaches and gives reasons why automated testing is preferable for our purposes.

- **Manual testing**. In this method, software testers execute test cases and generate test reports without using automated software testing tools. The greatest benefit of manual testing is its easy realization, as there is no need to write any code and all actions are performed by a human. Although this method provides great flexibility, it is a relatively high-cost approach, as some actions should be repeated several times. Therefore, the project tested manually is expensive. In conclusion, manual testing is time-consuming and expensive; moreover, it is not accurate due to human error at all times.
- **Automated testing**. Automated testing is performed by tools and scripts, so it is significantly faster than a manual approach. In contrast to the manual approach, this method is counted as reliable, as it eliminates the risk of human error. Moreover, this type of testing is more efficient, as it requires less time than the manual one.

Based on the mentioned facts, I decided to cover application with automated tests as it provides utmost efficiency and great reusability; moreover, it is cheaper in terms of required human resources.

9.1.2 Development and user testing

Testing can be performed during different development phases. The decision of whether tests are executed by end-users in the final phase of development or by programmers in the initial stage is based on the aims of testing. The gold rule of testing is that the earlier it is performed, the cheaper it is. Therefore, I concentrated my efforts on development tests that can be performed even in earlier stages of application creation. Although user testing has some benefits, it is not required for this development phase.

9.1.3 Tools

There are many tools for application testing, the choice is based on the performance and complexity requirements. The overview of used test types:

- **Static testing** does not involve program execution. These tests can be performed manually or using special software. A lot of services provide static code analysis, but since all CZM's applications use SonarQube, this solution is used in the e-mail service as well.
- **Unit tests** aim to verify behavior of the isolated code pieces. In this way, the code parts are tested independently and its results are not affected by each other.
- **API tests** intent to validate the execution of REST requests. These tests can be integrated into the CI/CD pipeline as well, which is especially useful for ensuring APIs compatibility.

9.2 Development testing

The e-mail service contains several modules, and all should be properly tested. The choice of test type depends on the module features. For example, business logic is generally covered by unit tests, whereas the data module is tested with integration tests.

9.2.1 Unit tests

I implemented unit tests without the use of `SpringApplicationContext`. It helps to reduce test execution time since Spring beans are not loaded. Unit tests cover the following modules:

- **Service module**. These tests generally verify error-free code execution during communication with the data layer. Due to the objective of writing independent tests, the repository beans are mocked. The tests cover successful and failed operations; therefore, there is validation of throwing checked exceptions.

- **REST module**. Unit tests for this layer were created using *MockHttpServletResponse*. An example of the test for the REST layer is presented below:

```
@Test
@DisplayName("addConfiguration returns 201 when configuration added")
void addConfigurationReturnsCreatedWhenConfigurationAdded()
throws Exception {
// when
MockHttpServletResponse response =
    mvc.perform(
            post("/configurations/")
                .contentType(MediaType.APPLICATION_JSON)
                .content(jsonConf.write(new Configuration()).getJson()))
        .andReturn()
        .getResponse();

// then
assertThat(response.getStatus()).isEqualTo(HttpStatus.CREATED.value());
}
```

9.3 Meeting acceptance criteria

CZM's analytics confirmed that the developed application meets the acceptance criteria. In addition, I conducted verification using the Postman API platform.

9.3.1 Test Scenarios

For testing purposes, I prepared the test scenarios that are presented in Table 9.1:

Scenario ID	Entity	Description
1	User	Administrator is able to add and remove users from the system
2	Configuration	Administrator is able to add and remove configuration from the system
3	Configuration	Administrator is able to grant access to the configuration for the particular user
4	Mailer	Administrator is able to add and remove mailer from the system
5	Mailer	Administrator is able to add a mailer to the configuration
6	E-mail	User is able to send an e-mail without explicit configuration
7	E-mail	User is able to send an e-mail with configuration
8	E-mail	User is able to send an e-mail with an attachment
9	E-mail	Administrator is able to view e-mails sent by the application
10	E-mail	User is able to view e-mails that they sent

Table 9.1: Test scenarios

9.3.2 Postman

In order to execute testing in Postman I prepared a collection of requests for each entity. Then, I wrote scripts using JavaScript for responses validation. As soon as the collection is run, Postman sends prepared requests and executes defined scripts. An example of a test that compares the data in the received response and the expected one is presented below:

```
pm.test("Returns all mails", function () {
    var expectedRes =
    [{
        "receivers": [
            "test@gmail.com"
        ],
        "sender": "czm-hub",
        "subject": "important",
        "body": "new",
        "date": "2022-04-22T17:51:22.000+00:00"
    },
    {
        "receivers": [
            "test@gmail.com",
            "example@yahoo.com"],
        "sender": "czm-camunda",
        "subject": "new service",
        "body": "We need it tomorrow!",
        "date": "2022-05-01T09:12:56.000+00:00"
    }]
    var jsonData = pm.response.json();
    console.log(jsonData);
    pm.expect(jsonData).to.deep.equal(expectedRes);
});
```

Other tests validate the response in the following way:

```
pm.test("Sends e-mail with conf", function () {
    pm.response.to.have.status(200);
});
```

All Postman tests were performed successfully. The results are presented in Figure 9.1:

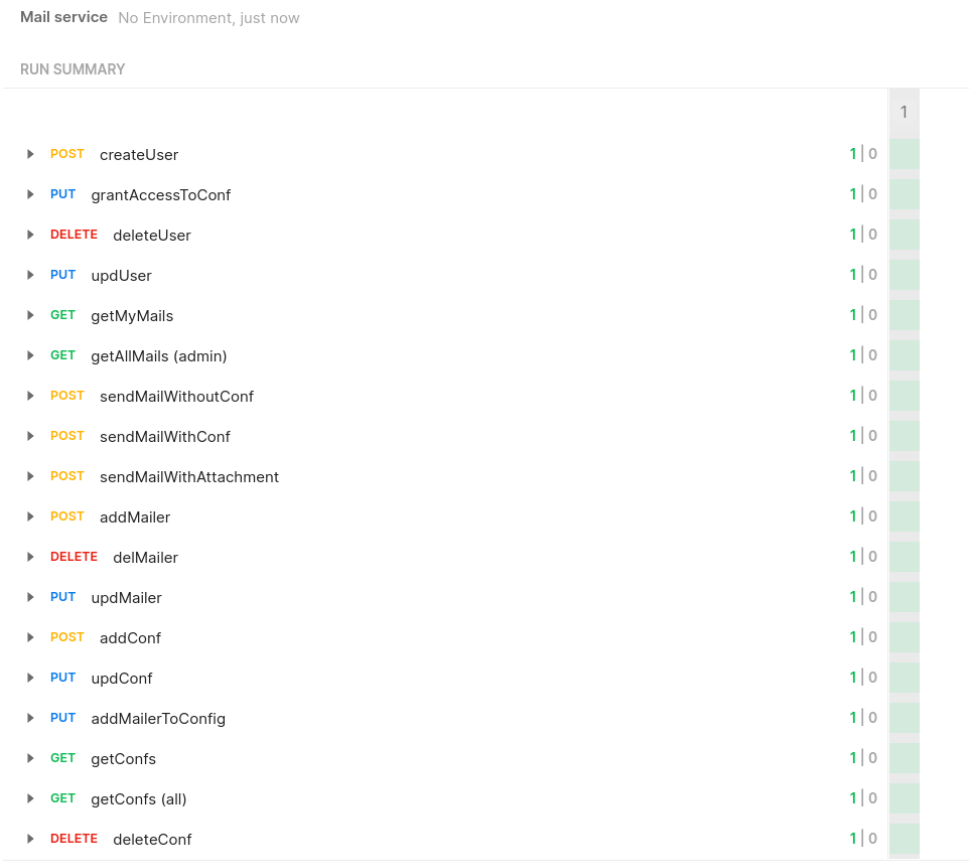

Figure 9.1: Results of the tests

9.4 Conclusion

The application was tested using various tools, such as JUnit, SonarQube, and Postman. In summary, 16 unit tests and 18 Postman tests were performed. During the testing, no drawbacks in terms of functionality were detected. Moreover, the application meets the acceptance criteria.

Chapter 10

Deployment and application maintenance

Once the application is developed and tested, it is time to prepare for deployment. This chapter contains details of the final development phases. First, I describe the deployment process and then provide information about the implementation of the Health Check module.

10.1 Docker

There are many deployment tools, but since the CZM's applications are running in the Docker Swarm, the e-mail application is also deployed there.

Docker is a container solution that enables us to construct isolated and consistent application environments with speed at scale. This approach is considered to be more lightweight, as containers do not contain operating system images. [47]

10.1.1 Different environments

In order to provide support for running different versions of applications simultaneously, I prepared test and production `docker-compose` files. These files not only contain information about infrastructure politics and used image, but also keep variables, specifying the actual environment. Part of `docker-compose-test.yml` is presented below:

```
version:...
services:
  mail_service:
    image: ...
    deploy:
      ...
    ports:
      ...
    environment:
      SPRING_PROFILES_ACTIVE: test
      SWARM_ENVIRONMENT: test
    secrets:
```

```
      - mail-service-mongodb-uri-test
      - mail-service-db-encrypt-test
      ...

secrets:
  mail-service-mongodb-uri-test:
    external: true
  mail-service-db-encrypt-test:
    external: true
    ...
```

Database

In order to establish a connection with the database, it is necessary to specify some credentials, for example, host, username, etc. Although, for development purposes, it is sufficient to set them directly to `application.properties` on the developer's machine, it is not an approach for the deployed application due to security concerns. Therefore, there is a need to keep these credentials as Docker secrets in the Docker Swarm. Once the application is deployed, the secrets should be processed and set to application.properties, so that Spring can process them and establish a connection to the database. `docker-compose.yml` contains the credentials for establishing.

CZM infrastructure contains a general implementation of `org.springframework.boot.env.EnvironmentPostProcessor`. I extended this abstract class to make it fill in `application.properties` with the secrets located in the docker container. Therefore, docker secrets are processed in the following way:

```
@Slf4j
class DockerSecretsProcessor extends AbstractDockerPropertiesLoader {
  private static final String DB_URL = "mail-service-mongodb-uri";
  private static final String DB_ENCRYPT = "mail-service-db-encrypt";
  ...

  @Override
  protected void loadProperties(
    ConfigurableEnvironment environment,
    SpringApplication application) {
    loadTokensAndCredentials(environment);
  }

  private void loadTokensAndCredentials
  (ConfigurableEnvironment env) {
    Properties props = new Properties();
    props.put("spring.data.mongodb.uri", loadDockerSecret(DB_URL));
    props.put("cz.cvut.fel.mail-db-encrypt-secret",
    loadDockerSecret(DB_ENCRYPT));
```

```
        ...
        env.getPropertySources().addFirst(
            new PropertiesPropertySource("dbProps", props));
    }
}
```

10.1.2 Logging from Docker container

Elasticsearch is set up to collect logs of the email service running in the docker swarm. These logs must then be handled by Kibana to provide the programmer with access to their visualization; in order to achieve it, the log format was specified in `logback.xml`. Thus, a programmer has a comprehensive overview of the events that occur inside the container.

10.2 GitLab Continuous Delivery

I set up the GitLab pipeline to ensure continuous delivery. The stages executed once the new code is pushed to the `test` or `production` branches are presented below:

1. Build. This stage creates the jar of the application.
2. SonarQube. This stage executes the code analysis.
3. Image. This stage builds an image and pushes it to the GitLab service registry.
4. Deploy. This stage creates and updates a stack from a compose file on the swarm.

10.3 The health check module

The e-mail service contains several parts, and all of them should run smoothly. The health check module allows one to verify whether all components of the application are available and running. Moreover, it allows one to predict the state of the system in the future by analyzing errors and disk space consumption. With this information, the development team can take measures, such as restarting instances or changing space strategy. This allows to minimize the perils that application will fall in the future.

Usually, the health check module is implemented by sending heartbeats to the service. Nevertheless, Spring Boot provides an out-of-box tool that is called Actuator. This solution provides several options; not only it checks status of the whole application, but also it can verifies connection to the database, space usage and much more.

The health check module is a widely used practice in CZM, therefore, I added the actuator to the e-mail service as well. For current purposes, verification of the application status is sufficient. It can be shown in the dashboard with all running applications that facilitate maintenance and reduce the time required to realize that the application is down.

Chapter 11

Conclusion

The chapter provides an overview of the thesis results and suggests ideas for further application enhancement.

11.1 Results

The primary purpose of the thesis was to develop an e-mail service for the customer. Not only was this goal successfully achieved, but several supporting tasks such as gaining theoretical knowledge about e-mail processing, performing analysis for a future solution, and designing the application were also performed. The development cycle also included testing and deployment phases. To conclude, all developmental milestones were successfully passed and the application entered the final stage of the development process, the maintenance phase.

Project goals are considered to be achieved according to their definitions. Therefore, the work results are counted as successful and satisfactory.

11.2 Maintenance and future development

The e-mail service uses convenient tools to facilitate maintenance, allowing the development command to take measures immediately if the application is down.

The developed application follows a modern design, built with modular and layered architectural patterns. As a result, the e-mail service is reusable and adaptable, as it can be efficiently modified without decreasing the quality of the existing product. Moreover, the application is covered with automatized tests that reduce the risk of destroying core functionality when implementing new features.

Based on these facts, adding new functionality to the e-mail service will not cause problems. Possible enhancements include extending the configuration functionality by adding new parameters, such as permission for attachments and displayed sender name. Moreover, e-mail templates can be added, so the user's input will be formatted in a special way before it is sent.

Appendices

Appendix A

Bibliography

1. RESNICK, Pete. *Internet Message Format* [RFC 5322]. RFC Editor, 2008. Request for Comments, no. 5322. Available from DOI: 10.17487/RFC5322.
2. SAINT-ANDRE, Peter; CROCKER, Dave; NOTTINGHAM, Mark. *Deprecating the "X-" Prefix and Similar Constructs in Application Protocols* [RFC 6648]. RFC Editor, 2012. Request for Comments, no. 6648. Available from DOI: 10.17487/RFC6648.
3. FREED, Ned; BORENSTEIN, Dr. Nathaniel S. *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies* [RFC 2045]. RFC Editor, 1996. Request for Comments, no. 2045. Available from DOI: 10.17487/RFC2045.
4. FREED, Ned; BORENSTEIN, Dr. Nathaniel S. *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types* [RFC 2046]. RFC Editor, 1996. Request for Comments, no. 2046. Available from DOI: 10.17487/RFC2046.
5. LEVINSON, Dr. Ed. *The MIME Multipart/Related Content-type* [RFC 2387]. RFC Editor, 1998. Request for Comments, no. 2387. Available from DOI: 10.17487/RFC2387.
6. TROOST, Rens; DORNER, Steve; MOORE, Keith. *Communicating Presentation Information in Internet Messages: The Content-Disposition Header Field* [RFC 2183]. RFC Editor, 1997. Request for Comments, no. 2183. Available from DOI: 10.17487/RFC2183.
7. *Mail routing and the domain system* [RFC 974]. RFC Editor, 1986. Request for Comments, no. 974. Available from DOI: 10.17487/RFC0974.
8. *Domain names - implementation and specification* [RFC 1035]. RFC Editor, 1987. Request for Comments, no. 1035. Available from DOI: 10.17487/RFC1035.
9. CHHABRA, Gurpal; PROFESSOR, Chhabra; SINGH, Dilpreet; PROFESSOR, Bajwa. Review of E-mail System, Security Protocols and Email Forensics. *International Journal of Computer Science Communication Networks*. 2015, vol. 5, pp. 201–211.

10. BOCHMANN, G.; SUNSHINE, C. Formal Methods in Communication Protocol Design. *IEEE Transactions on Communications*. 1980, vol. 28, no. 4, pp. 624–631. Available from DOI: 10.1109/TCOM.1980.1094685.

11. *Message Access Paradigms and Protocols* [online]. Washington Edu [visited on 2021-12-10]. Available from: http://staff.washington.edu/gray/papers/imap.vs.pop.html.

12. ROSE, Dr. Marshall T.; MYERS, John G. *Post Office Protocol - Version 3* [RFC 1939]. RFC Editor, 1996. Request for Comments, no. 1939. Available from DOI: 10.17487/RFC1939.

13. MELNIKOV, Alexey; LEIBA, Barry. *Internet Message Access Protocol (IMAP) - Version 4rev2* [RFC 9051]. RFC Editor, 2021. Request for Comments, no. 9051. Available from DOI: 10.17487/RFC9051.

14. DEKENS, Berend. Relations Between In-and Outbound Email Traffic. 2022.

15. KLENSIN, Dr. John C. *Simple Mail Transfer Protocol* [RFC 5321]. RFC Editor, 2008. Request for Comments, no. 5321. Available from DOI: 10.17487/RFC5321.

16. GELLENS, Randall; KLENSIN, Dr. John C. *Message Submission* [RFC 2476]. RFC Editor, 1998. Request for Comments, no. 2476. Available from DOI: 10.17487/RFC2476.

17. SCHRODER, Carla. *Linux Cookbook: Practical Advice for Linux System Administrators: The Science of Microfabrication*. O'Reilly Media, 2004.

18. HUTZLER, Carl; CROCKER, Dave; ALLMAN, Eric P.; RESNICK, Pete; FINCH, Tony. *Email Submission Operations: Access and Accountability Requirements* [RFC 5068]. RFC Editor, 2007. Request for Comments, no. 5068. Available from DOI: 10.17487/RFC5068.

19. KLENSIN, Dr. John C.; GELLENS, Randall. *Message Submission for Mail* [RFC 6409]. RFC Editor, 2011. Request for Comments, no. 6409. Available from DOI: 10.17487/RFC6409.

20. CROCKER, Dave. *Internet Mail Architecture* [RFC 5598]. RFC Editor, 2009. Request for Comments, no. 5598. Available from DOI: 10.17487/RFC5598.

21. *Email processing* [online]. Zeste De Savoir, 2021 [visited on 2021-12-12]. Available from: https://zestedesavoir.com/media/galleries/5382/ae0bdae3-d5dd-45f8-afc2-ad74dd6060be.png.

22. MYERS, John G. *Simple Authentication and Security Layer (SASL)* [RFC 2222]. RFC Editor, 1997. Request for Comments, no. 2222. Available from DOI: 10.17487/RFC2222.

23. ZEILENGA, Kurt; MELNIKOV, Alexey. *Simple Authentication and Security Layer (SASL)* [RFC 4422]. RFC Editor, 2006. Request for Comments, no. 4422. Available from DOI: 10.17487/RFC4422.

24. ZEILENGA, Kurt. *The PLAIN Simple Authentication and Security Layer (SASL) Mechanism* [RFC 4616]. RFC Editor, 2006. Request for Comments, no. 4616. Available from DOI: 10.17487/RFC4616.

25. CATOE, Randy; KRUMVIEDE, Paul; KLENSIN, Dr. John C. *IMAP/POP AUTHorize Extension for Simple Challenge/Response* [RFC 2195]. RFC Editor, 1997. Request for Comments, no. 2195. Available from DOI: 10.17487/RFC2195.

26. *OAuth 2.0 Mechanism* [online]. Gmail for developers, 2022 [visited on 2022-04-09]. Available from: https://developers.google.com/gmail/imap/xoauth2-protocol.

27. SIEMBORSKI, Rob; MELNIKOV, Alexey. *SMTP Service Extension for Authentication* [RFC 4954]. RFC Editor, 2007. Request for Comments, no. 4954. Available from DOI: 10.17487/RFC4954.

28. FREED, Ned. *SMTP Service Extension for Command Pipelining* [RFC 2920]. RFC Editor, 2000. Request for Comments, no. 2920. Available from DOI: 10.17487/RFC2920.

29. MASUI, Kenji; TOMOISHI, Masahiko; YONEZAKI, Naoki. Secure implementation method of POP before SMTP using a relay server with SSL protocol. *Electronics and Communications in Japan Part Iii-fundamental Electronic Science - ELECTRON COMMUN JPN III*. 2004, vol. 87, pp. 27–34. Available from DOI: 10.1002/ecjc.20105.

30. *How to Handle SMTP Authentication | Mailtrap Blog* [online]. Smtp-auth, 2019 [visited on 2022-01-14]. Available from: https://mailtrap.io/blog/smtp-auth/.

31. REUTER, Adrian; BOUDAOUD, Karima; WINCKLER, Marco; ABDELMAKSOUD, Ahmed; LEMRAZZEQ, Wadie. Secure Email - A Usability Study. *CoRR*. 2021, vol. abs/2110.06019. Available from arXiv: 2110.06019.

32. MOORE, Keith; NEWMAN, Chris. *Cleartext Considered Obsolete: Use of Transport Layer Security (TLS) for Email Submission and Access* [RFC 8314]. RFC Editor, 2018. Request for Comments, no. 8314. Available from DOI: 10.17487/RFC8314.

33. LAU, Robert. Email Basics. In: 2021, pp. 281–307. ISBN 978-1-4842-6959-6. Available from DOI: 10.1007/978-1-4842-6960-2_12.

34. PODDEBNIAK, Damian; ISING, Fabian; BÖCK, Hanno; SCHINZEL, Sebastian. Why TLS is better without STARTTLS: A Security Analysis of STARTTLS in the Email Context. In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2021, pp. 4365–4382. ISBN 978-1-939133-24-3. Available also from: https://www.usenix.org/conference/usenixsecurity21/presentation/poddebniak.

35. HOFFMAN, Paul E. *SMTP Service Extension for Secure SMTP over TLS* [RFC 2487]. RFC Editor, 1999. Request for Comments, no. 2487. Available from DOI: 10.17487/RFC2487.

36. *WHY STUDY AT CTU?* [online]. CTU [visited on 2022-05-15]. Available from: https://www.cvut.cz/en/why-study-at-ctu.

37. *TIOBE Index* [online]. TIOBE Software BV [visited on 2022-05-15]. Available from: https://www.tiobe.com/tiobe-index/.

38. *NoSQL vs SQL databases* [online]. Mongo db, 2022 [visited on 2022-03-23]. Available from: https://www.mongodb.com/nosql-explained/nosql-vs-sql.

39. NAYAK, A.; PORIYA, A.; POOJARY, Dikshay. Article: Type of nosql databases and its comparison with relational databases. *International Journal of Applied Information Systems*. 2013, vol. 5, pp. 16–19.

40. *Types of NoSQL Databases* [online]. MongoDB [visited on 2022-04-15]. Available from: https://www.mongodb.com/scale/types-of-nosql-databases.

41. *DB-Engines Ranking* [online]. Solid IT [visited on 2022-05-15]. Available from: https://db-engines.com/en/ranking/document+store.

42. *ObjectId* [online]. MongoDB, 2022 [visited on 2022-04-23]. Available from: https://www.mongodb.com/docs/manual/reference/method/ObjectId/#objectid.

43. *MongoDB Manual* [online]. MongoDB, 2022 [visited on 2022-04-22]. Available from: https://www.mongodb.com/docs/manual/reference/database-references/.

44. *Using Document References* [online]. Spring Data MongoDB - Reference Documentation, 2022 [visited on 2022-04-23]. Available from: https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/.

45. UPADHYAY, Nitin. SDMF: Systematic Decision-making Framework for Evaluation of Software Architecture. *Procedia Computer Science*. 2016, vol. 91, pp. 599–608. Available from DOI: 10.1016/j.procs.2016.07.151.

46. FOWLER, Martin; RICE, David; FOEMMEL, Matthew; HIEATT, Edward; MEE, Robert; STAFFORD, Randy. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

47. BHARDWAJ, Dr; CHALLA, Rama. Virtualization in Cloud Computing: Moving from Hypervisor to Containerization—A Survey. *Arabian Journal for Science and Engineering*. 2021, vol. 46. Available from DOI: 10.1007/s13369-021-05553-3.

Appendix B

List of Abbreviations

Bcc	Blind carbon copy
Cc	Carbon copy
CRUD	Create, Read, Update, Delete
CZM	Center for Knowledge Management, the customer who ordered the application
DTO	Data Transfer Object
ESMTP	Extended Simple Mail Transfer Protocol
FQDN	Fully qualified domain name
Gitlab CD	GitLab Continuous Delivery
IMAP	Internet Message Access Protocol
IMF	Internet Message Format
MDA	Message Deliver Agent
MIME	Multipurpose Internet Mail Extensions
MSA	Message Submission Agent
MTA	Message Transfer Agent
MUA	Message User Agent
MX RR	Mail Exchanger Resource Record
POP	Post Office Protocol
SASL	Simple Authentication and Security Layer
SMTP	Simple Mail Transfer Protocol
SSL	Secure Socket Layer
TLS	Transport Layer Security
TTL	Time to live