Degree Thesis, Åland University of Applied Sciences, Degree Programme in Information Technology

INTEGRATION OF MESSAGING SYSTEMS

Simon Flankkila, Johnny Sundman

2021:10

Publishing date: 12.05.2021
Academic Supervisor: Björn-Erik Zetterman

DEGREE THESIS
Åland University of Applied Sciences

Study program:	Information Technology
Author:	Simon Flankkila, Johnny Sundman
Title:	Integration of Messaging Systems
Academic Supervisor:	Björn-Erik Zetterman
Technical Supervisor:	Crosskey Banking Solutions

Abstract

This essay serves as documentation for the development of a messaging system for Crosskey Banking Solutions.
Our mission was to create a standalone application for message management.

The application was developed using Java, Spring, MyBatis and Swagger. The application uses Model View Controller and Inversion of Control.

The result was a REST API that a user can use for message management, which can easily be further developed.

Keywords

Spring, Java, REST, RESTful, API, MyBatis, Swagger

Serial number:	ISSN:	Language:	Number of pages:
2021:10	1458-1531	English	37 pages

Handed in:	Date of presentation:	Approved on:
04.05.2021	12.05.2021	12.05.2021

EXAMENSARBETE
Högskolan på Åland

Utbildningsprogram:	Informationsteknik
Författare:	Simon Flankkila, Johnny Sundman
Arbetets namn:	Integration av meddelandesystem
Handledare:	Björn-Erik Zetterman
Uppdragsgivare:	Crosskey Banking Solutions

Abstrakt

Denna uppsats fungerar som dokumentation för utvecklingen av ett meddelandesystem för Crosskey Banking Solutions.
Vårt uppdrag var att skapa en fristående applikation för meddelandehantering.

Applikationen utvecklades med hjälp av Java, Spring, MyBatis samt Swagger. Applikationen använder sig av Model View Controller och Inversion of Control.

Resultatet blev ett REST-API som en användare kan använda för meddelandehantering, som enkelt kan vidareutvecklas.

Nyckelord (sökord)

Spring, Java, REST, RESTful, API, MyBatis, Swagger

Högskolans serienummer:	ISSN:	Språk:	Sidantal:
2021:10	1458-1531	Engelska	37 sidor

Inlämningsdatum:	Presentationsdatum:	Datum för godkännande:
04.05.2021	12.05.2021	12.05.2021

Table of Contents

1. INTRODUCTION 6

1.1 Purpose 6

1.2 Method 7

1.3 Limitations 7

1.4 Definitions 8

2. PROGRAMMING PRINCIPLES AND PRACTICES 9

2.1 Data Transfer Object 9

2.2 Model-view-controller 9

2.3 Inversion of Control 10

2.4 Dependency Injection 10

2.5 REST 10

3. FRAMEWORKS AND TOOLS 12

3.1 Spring Framework 12

3.1.1 Spring application context and Spring beans 12

3.1.2 Dependency injection 12

3.2 MyBatis 14

3.3 Jenkins 15

3.4 Gradle 16

3.5 Swagger 16

3.5.1 Swagger API Description 17

4. INTEGRATION DEVELOPMENT 20

4.1 Preparation 20

4.1.1 Startup meetings 20

4.1.2 Planning 20

4.1.3 Architectural Changes 21

4.1.4 Development preparations 21

4.2 Development 23

4.2.1 Creation of a new project 23

4.2.2 Getting started with the implementation 24

4.2.3 Implementation of DTOs 25

4.2.4 Implementation of service, repository and controller 29

4.2.5 Testing 31

4.2.6 Extra about the development process 32

5. CONCLUSION 33

5.1 The result 33

5.2 Reflections 34

REFERENCES **36**

1. INTRODUCTION

1.1 Purpose

The purpose of this thesis is to describe the integration of a messaging system application. Details and issues about the implementation will be discussed in chapter 4 *Integration Development*.

The application was commissioned by our external supervisor Crosskey Banking Solutions[1] in order to ease the workflow for one of their client's, Ålandsbanken[2], electronic communication with their customers.

In order for Ålandsbanken's bank personnel to communicate with their customers, they had to use two different kinds of systems before our integration. The *Backoffice* system for directly communicating with their customers and the *Easit* system for forwarding the message to a specific responsible branch in the bank E.g. a customer sends a message about wanting to extend its line of credit to the bank. The bank's customer service personnel read the message in the Backoffice system and then replicate the message in the Easit system. From the Easit system the message can be sent to e.g. the credits branch, that deals with credits. The credits branch can act upon the message they see in the Easit system and directly communicate with the customer.

Our mission was to ease the use of multiple systems. Instead of communicating with the clients in the Backoffice system - every message should be sent directly to the Easit system using our integrated application. Therefore eliminating manual work in terms of replicating messages and introducing a direct way of communication.

---

[1] Crosskey Banking Solutions is a Finnish software company that provides financial solutions to financial institutions in the Nordic area.(*About Crosskey*, 2015)
[2] Ålandsbanken is a commercial bank operating in Finland, Åland and Sweden.(*Om Ålandsbanken*, 2015)

1.2 Method

The first step was to analyze how the system works today. We had several start-up meetings with Ålandsbanken and Easit to understand how the systems are used today and what improvements the new system could provide. Analysis of the old code provided us with a fundamental understanding of how the system operated today and whether we could reuse any code. It was decided that our application would be independent of other systems, which meant no reuse of code.

Pair programming was used in the beginning of the project to obtain a mutual understanding of how the system operated and because the work had not yet developed large enough for us to be able to work separately. We worked separately on the different sub-tasks after the foundation of the project was implemented, but we kept in close contact with each other so that all sub-tasks fit well together. When issues arose, we used pair programming or enlisted the assistance of Crosskey's other employees.

The application was written in Java and used frameworks such as Spring to manage the web flow and handle security, as well as MyBatis to fetch and save data. Intellij Idea was used as the development environment because it is the standard IDE used by Crosskey and was something we have used before.

1.3 Limitations

Even though the project is a standalone project, it still has to be implemented in a way so it follows our internal supervisor's standard frameworks and tools that are already in place for other in-house applications. We did not have to perform any extra research about new frameworks and development tools, since everything was already accessible. We were therefore restricted to settle with Crosskey's versions of e.g. Spring, Java and an older version of the persistence framework MyBatis.

All the data models that we used and created, had to be verified by the third party Easit to assure the communication between our backend application and their system synergized without complications.

This project did not include any form of GUI/UI development - backend only.

1.4 Definitions

The following is a list of common words and abbreviations used in this essay.

- **Backoffice:** Application for internet banks used to administer customers and their netbank services.
- **Backoffice user:** Bank personnel that have access to Crosskey's Backoffice application.
- **CI/CD:** Continuous integration and continuous delivery.
- **Easit:** Third party system that acts as a frontend interface for our messaging application. (*Produkt Easit GO*, 2019)
- **Jboss:** Application server runtime platform used for building, deploying, and hosting Java applications and services. (*Red Hat JBoss Enterprise Application Platform (JBoss EAP)*, 2021)
- **Jenkins:** Server software that helps to automate building, testing and deployment of applications. (*Jenkins*, 2021)
- **JIRA:** Software tool used by agile teams to keep track of developers workflow. (Atlassian, 2021)
- **JSON:** JavaScript Object Notation. A human readable file format used to store and transmit data objects. (*JSON*, 2021)
- **Mybatis:** A persistence framework with support for custom SQL and coupling to Java objects. (*Mybatis*, 2021)
- **Spring:** An application framework and inversion of control container for the Java platform with extensions useful for web development. (*Spring Makes Java Simple*, 2021)
- **Spring bean:** Java objects that are created and wired by the Spring container. Configurable from an xml file.
- **XML:** Extensible Markup Language. A universal markup language that specifies the encoding rules for a file that is both machine and human readable.

- **YAML:** YAML is a human friendly data serialization standard for all programming languages. (*The Official YAML Web Site*, 2021)

2. PROGRAMMING PRINCIPLES AND PRACTICES

2.1 Data Transfer Object

A data transfer object (DTO) is a simple class with only getters and setters and no business logic in it. DTOs are used to reduce the number of calls needed to a remote interface (e.g. web service). Instead of repeatedly calling the interface to aggregate the requested data, the interface can be called once with all data collected in a DTO (*Data Transfer Object*, 2020).

2.2 Model-view-controller

Model-view-controller (MVC) is one of the most popular software design patterns. The idea is to separate your program so that each separate component has one unique purpose. These separate components are called model, view and controller. Figure 1 shows an overview of these components.

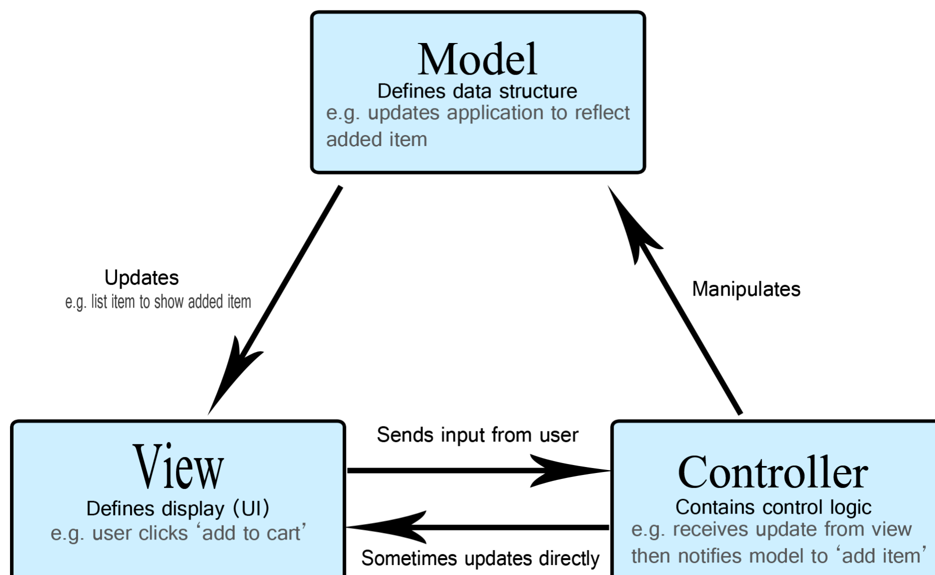

Figure 1. Model, View and Controller and how they interact with each other. (MVC - MDN Web Docs Glossary: Definitions of Web-Related Terms, 2021)

The model manages the data, logic and rules of the program. In our case this is the different data transfer objects that are used to hold our data, as well as our service, repository and database that we use to fetch, modify and save our data.

The view is what a user interacts with and displays data to the user. We did not do any development on the view in our project as that was done by Easit.

The controller handles all incoming requests and acts as an interface between the model and the view. In our project the REST-API acts as a controller. The controller receives calls from Easit's view who will then fetch or update the database via our services and repositories that is our model (*MVC - MDN Web Docs Glossary: Definitions of Web-Related Terms*, 2021).

2.3 Inversion of Control

Inversion of Control (IoC) is a programming principle that inverts the flow of control. This is accomplished by ensuring that a class is only aware of the interface that the implementation is supposed to fulfill, rather than the implementation itself. The IoC principle aids in the creation of loosely coupled classes that are easier to test, maintain, and extend (Crusoveanu, 2016).

2.4 Dependency Injection

Dependency Injection is a design pattern used to implement IoC. Dependency injection is done by providing the objects that a class needs via a constructor or setter instead of having the class construct them itself. This decoupling makes it easier to write tests for an application as well as changing its functionality by simply injecting a different implementation to a class. In our project we used Spring to handle the dependency injection via our Spring beans, which you can read more about in chapter 3.1 *Spring Framework* (*Dependency Injection*, 2021).

2.5 REST

Representational State Transfer (REST) is a software architectural style that facilitates communication between systems by providing a set of rules that an application must obey.

Applications that follow these set of rules are often called RESTful systems.

One of the most important rules to follow is statelessness. A system is stateless if communication between client and server can be done without knowing the state of each other. In other words, both the client and server can understand the data between them without having to know about any previous data or communication.

In a REST architecture, clients submit requests for resources to be retrieved or modified, and the server responds to these requests. The requests can be made up of a HTTP method, a message body containing data, and a URL to uniquely identify a resource. There are four basic HTTP methods:

- GET - For retrieving a resource
- POST - For creating a new resource
- PUT - For updating a resource
- DELETE - For removing a resource

The message body is usually made up of HTML, XML, or as in our case JSON. The URL can look something like *http://api.example.com/messages* where *messages* is the resource we want to create, retrieve or modify.

When the server receives a request from the client it will try to fulfill the request and respond with a response code, and in some cases also a body containing the requested data. The most common response codes are the following (*What Is REST?*, 2021):

- 200 (OK) - The HTTP request was successful
- 201 (CREATED) - The resource was successfully created
- 400 (BAD REQUEST) - The request cannot be processed because of bad request syntax, excessive size, or another client error.
- 404 (NOT FOUND) - The resource could not be found
- 500 (INTERNAL SERVER ERROR) - A generic answer for an unexpected failure

One of the benefits with developing RESTful applications is that development can start on the client and server side at almost the same time. But to do this we need a good way to show the contracts and functionality of our REST-API. In our project we used Swagger as a way to

illustrate our REST-API, which you can read more about in chapter 3.5 *Swagger* (*What Is REST?*, 2021).

3. FRAMEWORKS AND TOOLS

3.1 Spring Framework

Spring is an open source framework, and the main goal of the framework is to make it easy to create Java enterprise applications. The framework offers the flexibility to create many kinds of architectures and supports a wide selection of application scenarios (*Spring Framework Overview*, 2021).

3.1.1 Spring application context and Spring beans

Applications are composed of many components, and each component is responsible for its own piece of application functionality. Every component has to coordinate with other application elements to get the job done. When the application is run, these components need to be created and introduced to each other. At its core, Spring offers functionality that creates and manages application components. This functionality is called a *container*, often referred to as the *Spring application context.* The mentioned components are called *beans*, and they are wired together inside the Spring application context to make a complete application (Walls, 2019).

3.1.2 Dependency injection

Wiring beans together is based on a pattern called *dependency injection*. Instead of having components create and maintain the lifecycle of other beans that they depend on, a dependency-injected application relies on a separate entity (the container) to create and maintain all components and inject those into the beans that need them (Walls, 2019).
The examples in figure 2 are inspired by Craig Walls' examples from Spring In Action (2019).

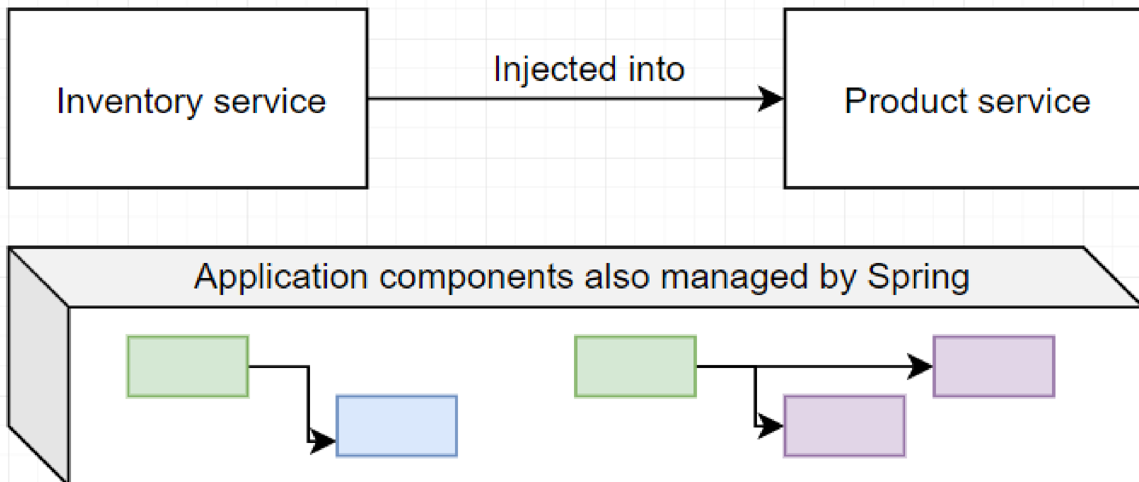

Figure 2. The Spring application context managing and injecting components into each other.

Figure 2 illustrates an example of dependency injection. The product service depends on the inventory service to be able to provide a complete set of information about products. To guide Spring's application context to wire beans together, is done either with XML files describing the components and their relations to other components, or a Java-based configuration.

In Figures 3 and 4 are two examples of bean mappings and the result is equivalent to each other. One XML configuration and one Java-based configuration.

```
<bean id="inventoryService"
        class="com.example.InventoryService" />

<bean id="productService"
        class="com.example.ProductService"
    <constructor-arg ref="inventoryService" />
</bean>
```

Figure 3. Bean configuration in XML.

```java
@Configuration
public class ServiceConfiguration {
    @Bean
    public InventoryService inventoryService() {
        return new InventoryService();
    }

    @Bean
    public ProductService productService() {
        return new ProductService(inventoryService());
    }
}
```

Figure 4. Bean configuration in Java.

3.2 MyBatis

MyBatis is a persistence framework with support for stored procedures, custom SQL and advanced mappings. MyBatis aims to automate mappings between Java objects and SQL databases (MyBatis, 2021). See the example in figure 5 of how the mapping is done in code:

```xml
<resultMap id="customerResultMap" type="Customer">
    <id property="id" column="id" />
    <result property="name" column="name"/>
    <result property="dob" column="dob"/>
</resultMap>
```

Figure 5. An example of a resultmap of type Customer. Customer represents a Java object.

```xml
<insert id="insertCustomer" parameterType="java.util.Map">
  insert into Customer (name, dob, id)
  values (#{name},#{dob}, #{id})
</insert>

<update id="updateCustomer" parameterType="java.util.Map">
  update Customer set
    name = #{name},
    dob = #{dob}
  where id = #{id}
</update>

<delete id="deleteCustomer" parameterType="java.lang.String">
  delete from Customer where id = #{id}
</delete>

<select id="selectCustomers" parameterType="java.lang.String"
  resultMap="customerResultMap">
    select id, name, dob
    from example_table
    where id = #{id}
</select>
```

Figure 6. Example of an insert, an update, a delete and a select operation. The select operation returns the specified resultMap - customerResultMap

As one can see, the mapping is done by writing standard SQL statements in XML, i.e. *select*, *insert*, *update* or *delete* operations. Result maps are recommended since they map the result from the database into Java objects.

3.3 Jenkins

Jenkins is an open source automation server and offers a simple way to set up continuous integration and continuous delivery (CI/CD) environment for different combinations of programming languages and source code repositories. Jenkins gives a robust way to integrate the entire chain of build, test and development tools (Heller, 2020).

Regarding our application, the only Jenkins configuration we had to perform was to add an organization wide predefined Jenkins-file to our application. The file contains information about what in-house pipeline configuration our application should use once in the actual pipeline flow to work accordingly. This is due to Jenkins being a company standard at Crosskey. No Jenkins pipeline configuration was needed on our behalf since it was already configured by our supervisor.

3.4 Gradle

Gradle is a build automation tool written in Java, Groovy and Kotlin. Gradle automates the compiling, testing, packaging, and deployment of software or other types or projects (Klein Ikkink, 2012). Gradle's core model is based on tasks. That means that a build configures a set of tasks and wires them together based on their dependencies and determines which tasks that need to be run in which order, and then proceeds to execute them (*What Is Gradle?*, 2021). See figure 7 below.

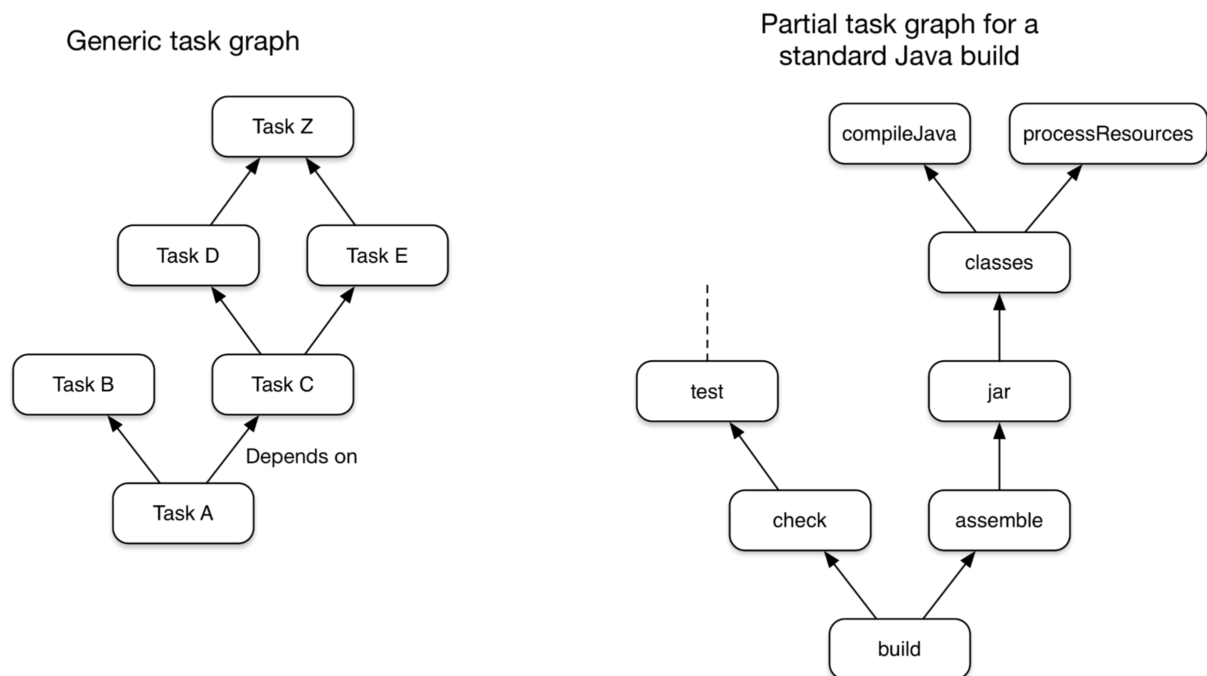

Figure 7. Gradle task graphs by Gradle Inc 2021 (CC BY-NC-SA 4.0).

3.5 Swagger

Swagger is a powerful set of API developer tools that enable development across the entire API lifecycle - design, documentation, testing and development. Swagger consists of a mix of

tools, both open source and commercially available. These tools allow developers to describe the structure of their APIs so that both machines and humans can read them. The description of the API structure can either be created by annotating the source code with Swagger annotations and generating the API description[3] from it, or by directly writing a configuration describing the API in JSON or YAML. The API description configuration needs to contain information about what operations that the API is going to support, what API parameters are needed, what the API returns, and what authorization the API needs (*What Is Swagger*, 2021).

In our project we went with annotating the source code and then generating the API description.

3.5.1 Swagger API Description

Figure 8 is an example of how a Swagger API description could look like in YAML. Below are explanations from Swagger's official documentation (*What Is Swagger*, 2021).

[3] The generated API description is in YAML format.

```yaml
swagger: '2.0'
info:
  description: Example API documentation
  version: '1.0'
  title: Api Documentation
host: 'localhost:9999'
tags:
  - name: example-controller
    description: Example Controller
paths:
  '/example/api/customer/{name}':
    get:
      tags:
        - example-controller
      summary: getCustomer
      operationId: customerUsingGET
      produces:
        - '*/*'
      parameters:
        - name: name
          in: path
          description: name
          required: true
          type: string
      responses:
        '200':
          description: OK
          schema:
            $ref: '#/definitions/CustomerDTO'
        '401':
          description: Unauthorized
definitions:
  CustomerDTO:
    type: object
    properties:
      Name:
        type: string
      Age:
        type: string
```

Figure 8. A simple example of a Swagger API description

1. *swagger: '2.0'*

   Information about the current Swagger version.

2. *info*

   A short description of the API, information about the current API version and the title of the API

3. *host*

   Where the API can be reached. In this example, the API is reachable locally on port 9999.

4. *tags*

   Tags group operations for easier navigation. In this example we have one tag with one operation. In a real world scenario, there are many different tags with associated operations.

5. *paths*

   Contain information about the actual endpoint, in this example, an individual GET operation with the path '*/example/api/customer/{name}*'. Note that the paths configuration contains information about tags mentioned in the paragraph above. Doing it like this links this operation with the operation group '*example-controller*'. Summary and operationId are two optional ways to describe the operation in more detail. Produces (and consumes) define the MIME types[4] supported by the API. Operations can have parameters. The parameters can be passed via query string (/example/api/customer?name=bob), headers (X-ExampleHeader: Value), request body and like in our example in the URL path (/example/api/customer/{name}). In our example we specify that the parameter is in the path, it is a string and it is required. Responses define how the API should respond for each operation (a status code and an optional configured response body). In our example the API responses are status code 200 and 401. The 200 response contains a response body schema. The schema indicates what the response should contain. More about schemas in the next paragraph.

6. *definitions*

   Defines common data structures used within the API. They can be referenced via '*$ref*'. In the example, the response references the '*CustomerDTO'* object in the response block under schema.

[4] "A media type (also known as a Multipurpose Internet Mail Extensions or MIME type) is a standard that indicates the nature and format of a document, file, or assortment of bytes."(*MIME Types (IANA Media Types)*, 2021)

4. INTEGRATION DEVELOPMENT

4.1 Preparation

4.1.1 Startup meetings

About a year before we were informed that we would develop this project, our system architect together with the product owner held a workshop with Easit and the client, Ålandsbanken. We ourselves did not attend this meeting, but it was here that the basis for the project was planned. The system architect sketched a class diagram to get an overview of which systems would be included, as well as sequence diagrams to show the interaction between the subsystems.

Before we began the development of the application, we got to attend a meeting where the system architect, product owner, Easit, and Ålandsbanken participated. During this meeting, it was discussed and checked to see if everyone was on the same page about how the system would work, as well as discussing schedules and deadlines in general. Later we also had a meeting with just the system architect to talk about DTOs, which systems would be included, and which database tables would be used.

4.1.2 Planning

We planned our work based on the outcomes of our meeting with the system architect. By writing JIRA stories that described what needed to be done, we were able to break the work down into smaller chunks. We attempted to divide the project into sections that could be completed in parallel, and thus reducing the amount of time we had to wait for one another. For this reason, we collaborated when writing many of the stories. JIRA was also used to keep track of the development's progress and who would be responsible for what.

4.1.3 Architectural Changes

Our initial idea[5] of how to implement the third party integration, and the idea we first embraced and went with, was to integrate the functionality in the already existing netbank application. The functionality would provide communication between the netbank and Easit's system by consuming an API provided and developed by Easit. After a meeting with Easit, it was made clear that we were not implementing the functionality like they expected it to be. Easit was expecting an API providing them with message data from our databases instead of them providing the netbank with data.

We had to roll back our progress and go with the new approach creating a standalone application providing Easit with data using an API. This thesis is about the new and correct approach.

4.1.4 Development preparations

Based on our startup meeting with the system architect, we decided to develop a completely independent application to reduce dependence on other and old systems to hopefully reduce development time, as there was little to no code we could reuse in the older systems. This also allowed us to have more flexibility in the development of the application. It was decided that the application would be a REST API that Easit could send requests to. During the development, Git and Bitbucket were used for version control. The system architect created a new repository in Bitbucket for us that only contained our code. Jenkins was also configured for multibranch CI/CD by the system architect.

Multiple diagrams were created to illustrate how the different components would work together. Figure 9 illustrates how our new Internal-messaging-api was integrated with the same netbank database as the already existing Netbank and Json API applications. Our new *internal-messaging-api* was completely separate from all Netbank and Json API logic, and has only the database in common. Figure 10 shows how an end user can create a new message with the already existing mobile application and have it saved in our common database. When Easit calls our GET /messages endpoint we fetch all messages that have not yet been previously sent to Easit and include them in our response. To avoid sending the

[5] Based on outdated workshop documents and diagrams.

same messages over and over again, we use a flag in our database table to mark them as exported. This way we can make sure the messages are only sent once. Figure 11 illustrates how a back office personnel can reply to messages. All messages sent to Easit have a *threadId* included in the response. This makes it possible for personnel to respond to specific messages. As soon as a back office personnel has sent the message it is readable by the end user.

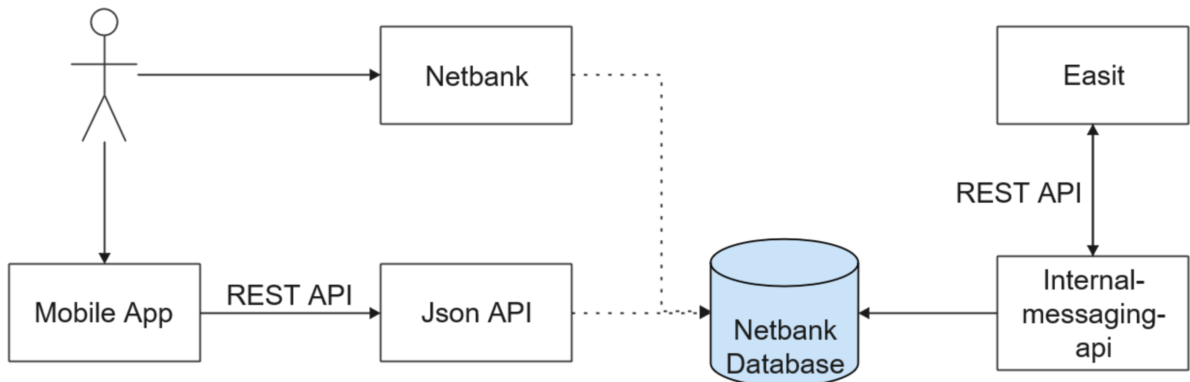

Figure 9. Diagram showing the interaction between the different components

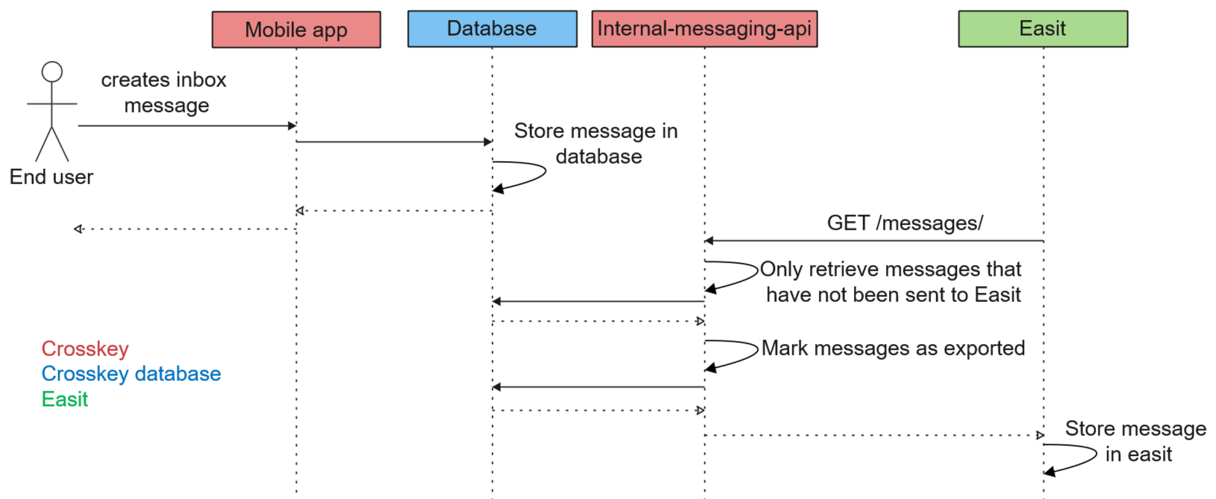

Figure 10. Sequence diagram showing the creation and fetching of messages

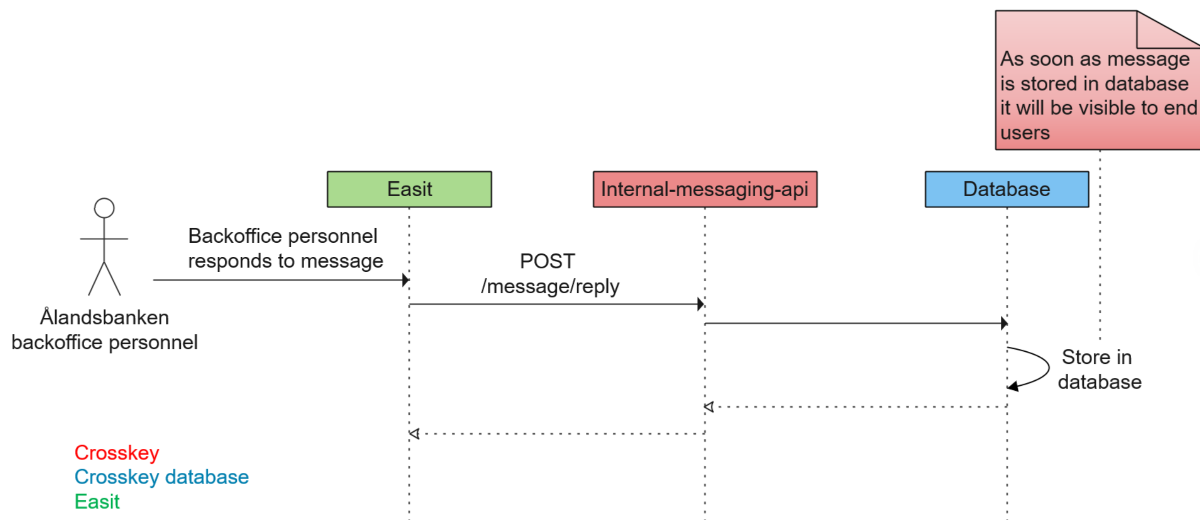

Figure 11. Sequence diagram showing how the back office personnel can reply to customers messages

4.2 Development

4.2.1 Creation of a new project

In the beginning of the implementation process we started off with mapping out how we wanted the application structure to be. We looked at already existing applications in the organization to find a common structure practice. The common practice when it comes to standalone API applications is to use Gradle as a build automation tool, Spring to handle the web communication, the programming language Java version 8, Jenkins for CI/CD, MyBatis for repository mappings, and Swagger for communicating and testing the API. These tools were more or less required to achieve our goal.

The application was developed using the IDE IntelliJ Idea. IntelliJ provides the functionality to create different new projects supporting different languages and build tools. We chose to create a Java 8 Gradle project named *internal-messaging-api*. IntelliJ aided with the creation of a complete project structure with needed files. To the project root *internal-messaging-api* we added a new module called *internal-messaging-api-web*. This module would contain all the source code of the application.

We added a directory called WEB-INF to the new module to be able to run the application like a web application. The WEB-INF directory contains a couple of files, but the two most

important are the Spring application context configuration (*applicationContext.xml*) and definitions about the application that the server needs to know about (*web.xml*).
To be able to run our application in a CI/CD pipeline we added a Jenkinsfile to the project root that defines the Jenkins pipeline. In our case, we imported a company standard configuration pointing to a shared Jenkins library that loads when inside the actual pipeline.

After all this configuration, we could run the Gradle build task that assembles, tests and builds the entire application without any errors.

4.2.2 Getting started with the implementation

During the development of the application we needed a way to deploy our application and test it locally. For this a virtual machine running Linux and Jboss was used. The virtual machine resembles the production environment and its Jenkins pipeline. This way we could deploy our application to the local virtual machine and test it locally.

We confirmed that the application deployed successfully and started the implementation of the Swagger UI[6]. To get the Swagger UI working we needed to create a Java class containing the configuration of Swagger. Figure 12 depicts what was needed for our configuration class. Most of the configurations were made up of annotations. @Configuration tells Spring that the class contains one or more bean methods (*Configuration (Spring Framework 5.3.6 API)*, 2021). @EnableSwagger2 Indicates that Swagger support should be enabled (*EnableSwagger2*, 2017). @PropertySource tells Spring a properties file should be used and where that file is located (*PropertySource (Spring Framework 5.3.6 API)*, 2021). After this configuration class was in place it was possible to deploy the application and browse to *http://bips-devbox:3000/messaging-api/swagger-ui.html* and see an empty Swagger UI.

[6] Swagger UI is a user interface used to visualize and interact with the API's resources (*REST API Documentation Tool*, 2021)

```java
@Configuration
@EnableSwagger2
@PropertyResource("classpath:swagger.properties")
public class ApplicationSwaggerConfig {

    @Bean
    public Docket swaggerSpringMvcPlugin() { ... }
}
```

Figure 12. Configuration of class to enable Swagger UI

Because this application was to be deployed for both the Finnish and Swedish parts of Ålandbanken we needed a way to easily configure our application for both use cases. Luckily this was done automatically on the test, stage and production servers by including arguments to the Gradle task. When deploying the application locally however we needed to change the server name and port number. This needs to be done manually every time you want to switch between the Finnish and Swedish deployments by editing Gradle's properties file. In our Gradle properties file we simply mark the configuration we don't want to be active as a comment. This makes it fast and easy for anyone to switch configurations.

4.2.3 Implementation of DTOs

Verifying that the newly created project was reachable locally from Swagger UI we could start implementing our DTOs. The DTOs contain information about the different requests and responses that our API would consume and/or produce. We had documentation from Easit stating how they expected certain models to behave and what information to contain. We created the following DTOs:

- *CreateMessageRequestDTO*
  Represents the actual message and message information to be sent within a request body. Used when creating a new message.
- *MessageResponseDTO*
  Represents a message fetched from the database. Returned within a response body when fetching messages.

- *ReplyMessageRequestDTO*
  Represents a message and message information to be sent within a request body when replying to a message.
- *AttachmentDTO*
  Represents an attachment. Optional when sending a message. This DTO is a private field in all the other DTOs.

Figure 13 depicts the source code for CreateMessageRequestDTO. Notice the Swagger annotations for the different fields. All the DTOs are implemented in the same fashion.

When the DTOs were in place we created a controller class - *MessagesController*. This controller would, as we mentioned earlier in 2.2 *Model-view-controller* , receive calls from Easit's view who will then fetch or update the database.

Our controller class has three crucial annotations that are required for our application to work properly. The first annotation is @Controller that tells the spring application that this class is a component that needs to be automatically mapped in the Spring application context and also be handled by Spring. Otherwise Spring would not understand the business/representation layer purpose of the class. The second annotation is @RequestMapping(/internal-messaging). This annotation, simply put, maps web requests to Spring controller methods, and in this case - also classes. By annotating the whole class we map the specific request path */internal-messaging* onto the controller (Thompson, 2017). The last and third annotation @Api tells Swagger that the controller class is a Swagger resource and needs to be generated automatically to YAML format (to correctly be displayed in Swagger UI).

```java
public class CreateMessageRequestDTO {
    @ApiModelProperty(notes = "subject", required = true)
    private String subject;

    @ApiModelProperty(notes = "content")
    private String content;

    @ApiModelProperty(notes = "customer information (ssn)", required = "true")
    private String ssn;

    @ApiModelProperty(notes = "list of attachments")
    private List<AttachmentDTO> attachments;

    @ApiModelProperty(notes = "responder (replier)")
    private String responder;

    @ApiModelProperty(hidden = true)
    private String messageId;

    @ApiModelProperty(hidden = true)
    private String helpdeskId;
}
```

Figure 13. CreateMessageRequestDTO with Swagger annotated fields. The ApiModelProperty annotation specifies what information to display in Swagger UI. For example - "notes = 'customer information (ssn)'".

With this configuration mapped, we created three methods in the controller class:

- *CreateMessageResponseDTO createMessage(CreateMessageRequestDTO dto)*
  Method for creating a new message. Takes a CreateMessageRequestDTO as parameter (request body) and returns a CreateMessageResponseDTO as a response.
- *List<MessageResponseDTO> getMessages(Date fromDate, Date toDate)*
  Method for fetching messages. Takes two optional parameters for fetching messages between specified dates. Returns a list of MessageResponseDTOs.
- *ReplyMessageResponseDTO replyMessage(String threadId, ReplyMessageRequestDTO dto)*
  Method for replying on a message. Takes two required parameters, threadId specifies what thread/message to reply to, and the actual message - ReplyMessageRequestDTO.

As one can see, all these methods represent the functionality of the API. For Easit to be able to populate their view with data these methods/operations (create a message, fetch messages and reply on an existing message) were needed. We are going to walk through and explain one of these methods since they are all implemented in a similar fashion, and we cover almost everything, from annotations to return values, by explaining one of them.

Figure 14 depicts the method to create a message. A quick explanation of the annotations: @ApiOperation is a method level description of the Swagger method. The parameter *value* describes the method, *note* is simply an additional description about the method and *response* explains what type the response should be. CreateMessageResponseDTO does only contain a single field with information about a thread id. The thread id is the id of the new message. This information will be displayed in the Swagger UI. @PostMapping maps HTTP POST requests onto specific handler methods. We specify the value */message*, meaning from base path */internal-messaging*, this method is reachable from */internal-messaging/message*. The *consume* and *produce* parameters explain what the application is supposed to generate (produce) and what kind of input format the application expects (consume). We specified application/json for both the parameters. @ResponseBody tells the controller method that the return value is automatically serialized into JSON (Baeldung, 2021).The last annotation @ResponseStatus specifies the response status of the controller method. Normally when a controller method returns successfully it returns HTTP 200 (OK). In our case we wanted it to return HTTP 201 (CREATED) to indicate that a message had been created.

The method *createMessage* returns, as already mentioned, a DTO containing a single field representing the ID of the new message (thread). The parameter of the method does also have an annotation - @RequestBody. This annotation maps the request body to a transfer or domain object, enabling automatic deserialization of the incoming body onto a Java object (Baeldung, 2017). The Java object in our case is a CreateMessageRequestDTO object (see figure 13).

```
@ApiOperation(value = "Create message", notes = "Create message",
        response = CreateMessageResponseDTO.class)
@PostMapping(value = "/message", consumes = MediaType.APPLICATION_JSON_VALUE,
        produces = MediaType.APPLICATION_JSON_VALUE)
@ResponseBody
@ResponseStatus(code = HttpStatus.CREATED)
CreateMessageResponseDTO createMessage(@RequestBody final CreateMessageRequestDTO dto) {
        //logic here
}
```

Figure 14. The controller method "createMessage". The logic is left out here since it will be introduced and explained later in chapter 4.

4.2.4 Implementation of service, repository and controller

```
<resultMap id="messageResponseDTO" type="fi.crosskey.internal.messaging.domain.model.dto.MessageResponseDTO">
    <result property="threadId" column="threadId"/>
    <result property="messageId" column="messageId"/>
    <result property="messageNumber" column="messageNumber"/>
    <result property="subject" column="subject"/>
    <result property="content" column="content"/>
    <result property="customerInformation" column="customerInformation"/>
    <result property="category" column="category"/>
    <result property="resultOffice" column="resultOffice"/>
    <result property="customerAdvisor" column="customerAdvisor"/>
    <result property="locale" column="locale" javaType="fi.crosskey.internal.messaging.domain.model.Locale"/>
    <result property="messageSender" column="messageSender" javaType="fi.crosskey.internal.messaging.domain.model.MessageSender"/>
    <result property="identification" column="identification"/>
    <result property="created" column="created"/>
    <collection property="attachments" ofType="fi.crosskey.internal.messaging.domain.model.dto.AttachmentDTO">
        <result property="fileContent" column="fileContent" jdbcType="BLOB"
        typeHandler="fi.crosskey.internal.messaging.infrastructure.BlobTypeHandler"/>
        <result property="fileName" column="fileName"/>
    </collection>
</resultMap>
```

Figure 15. Our MyBatis result map that is used to map database data to the java object MessageResponseDTO.

When the system architect and Easit had approved our endpoints, DTOs, and the overall functionality we started the implementation of our service and repository layers. The repository is responsible for all database operations. The repository maps the data to objects the database can handle. We used MyBatis for our repository. With MyBatis we were able to map all our operations and models in custom xml-files, an example of a MyBatis mapper can be seen in figure 15. The service layer is responsible for directing all calls to the correct repository methods.

We injected the service layer using dependency injection into the controller like figure 16 depicts. The annotation @Autowired allows Spring to resolve and inject collaborating beans

(the MessagesService bean) into the MessagesController (bean). This allows us to call the different service methods.

```
@Controller
@RequestMapping("/internal-messaging")
@Api(description = "Message", value = "message")
public class MessagesController {

    private final MessageService messageService;

    @Autowired
    public MessagesController(final MessageService messageService) {
        this.messageService = messageService;
    }

    //Annotations omitted for simplicity
    CreateMessageResponseDTO createMessage(@RequestBody final CreateMessageRequestDTO dto) {
            return messageService.createMessage(dto);
    }

    //..replyMessage, getMessages

}
```

Figure 16. Dependency injection via the MessagesController constructor.

Our dataflow is easiest explained with an example: Easit calls our POST */internal-messaging/message/* endpoint containing a social security number(SSN) as well as message details. The controller receives this call and sends it to the service. The service then first calls the repository to check if a user with that SSN exists. If the SSN does not exist the repository will throw a custom UserNotFoundException. The exception is then caught in the controller that will respond with a HTTP 404 ‘user not found’ error message. If a user with the given SSN exists however the service method will continue and call the repository that will save the message to the database.

The database table we used was already in place, but we needed to modify the fields a bit. We added the field *EXPORTED* that can either be 0 or 1, and this is used to indicate if a message has previously been sent to Easit or not. We also added a field *EXPORTED_TIME* that contains the date and time the message was sent to Easit. This was needed to keep track of when the messages had been sent. These two fields allow us to only send messages that had not previously been sent to Easit. But this also made local testing difficult, because only one request could be made before all messages were marked as exported and would therefore not

be fetched the next time a request was made. We solved this issue by utilizing Spring profiles. By checking if Spring profile 'dev' was active we could skip marking the messages as exported. This helped tremendously when developing and testing the application locally.

If the standard MyBatis mappings do not meet your needs, MyBatis offers an interface through which you can build your own typehandlers. We needed to map some strings to enums and back, so we created our own typehandler to handle these mappings. Figure 17 depicts one of our implementations of such a typehandler. Our typehandler converts the locale string MyBatis fetches from the database and converts it to our custom java-enum Locale.

```
/** Type handler for Locale */
public class LocalTypeHandler extends BaseTypeHandler<Locale> {

    @Override
    public void setNonNullParameter(final PreparedStatement ps, final int i,
                final Locale parameter, final JdbcType jdbcType) throws SQLException {
        ps.setString(i, parameter.getValue());
    }

    @Override
    public Locale getNullableResult(final ResultSet rs, final String columnName) throws SQLException {
        return getLocale(rs.getString(columnName));
    }

    @Override
    public Locale getNullableResult(final ResultSet rs, final int columnIndex) throws SQLException {
        return getLocale(rs.getString(columnIndex));
    }

    @Override
    public Locale getNullableResult(final CallableStatement cs, final int columnIndex) throws SQLException {
        return getLocale(cs.getString(columnIndex));
    }

    private Locale getLocale(final String value) {
        if (value == null) {
            return null;
        }
        else {
            return Locale.findByValue(value);
        }
    }
}
```

Figure 17. Our custom Mybatis typehandler

4.2.5 Testing

We created unit tests for all of our controller methods. Since we have already treated the controller method *createMessage,* we are going to show an example of the method testing *createMessage* (see figure 18). The method checks if the endpoint

/internal-messaging/message is reachable, if it is possible to do a POST request with required parameters (CreateMessageRequestDTO), if the request returns the expected HTTP status code (201) and the result from the POST request is not null and it contains some data.

All of our controller method tests have the same functionality, but with different values depending on what operation is to be tested.

```
@Test
public void createMessageTest() throws Exception {
    String requestBody = ow.writeValueAsString(new CreateMessageRequestDTO());

    MvcResult result = mockMvc.perform(post("/internal-messaging/message")
            .contentType(MediaType.APPLICATION_JSON_VALUE)
            .content(requestBody))
            .andExpect(MockMvcResultMatchers.status().isCreated())
            .andReturn();

    Assert.assertNotNull(result);
    Assert.assertNotNull(result.getResponse().getContentAsString());

}
```

Figure 18. Method testing the controller method "createMessage".

4.2.6 Extra about the development process

We have been working on this project for approximately 7 weeks. During this time we have been able to put in 10-15 hours per week on this project, since it has been parallel with school. During the development we also have had bi-weekly meetings with the product owner and system architect to ensure the work has progressed as expected. We also have kept close contact with the system architect for any question or problems that have come up, and also some general guidance.

5. CONCLUSION

5.1 The result

All our work resulted in a working REST API that has the functionality to fetch, reply to, and create messages. The API can also easily be tested from its Swagger UI, which can be seen in figure 19 and 20. All this will make it possible for Easit to call our endpoints and populate the view that they will develop. Our project is right now being run on an internal test server within Crosskey that any employee can run and test. Right now the plan is to push it to production in October. Our work can be seen as done as long as no issues arise with our implementation.

swagger
default (/v2/api-docs)
Explore

Internal Message API
RESTful API exposing Internal messaging functionality.

Created by Team Authentication, BIPSTeamAuthentication@crosskey.fi
Copyright (c) Crosskey Banking Solutions. All rights reserved.

messages-controller : Message
Show/Hide | List Operations | Expand Operations
POST /internal-messaging/message Create message
POST /internal-messaging/message/{threadId} Reply message
GET /internal-messaging/messages Get messages

[BASE URL: /messaging-api , API VERSION: master-SNAPSHOT]

Figure 19. Swagger UI showing our endpoints

messages-controller : Message   Show/Hide | List Operations | Expand Operations

POST /internal-messaging/message   Create message

Implementation Notes

Create message

Response Class (Status 201)

Created

Model | Example Value

CreateMessageResponseDTO {
threadId (string, *optional*): ID of the thread
}

Response Content Type application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
createMessageRequestDTO	(JSON body below)	createMessageRequestDTO	body	Model | Example Value


```
{
  "attachments": [
    {
      "fileContent": "string",
      "fileName": "string"
    }
  ],
  "content": "string",
  "responder": "string",
  "ssn": "string",
  "subject": "string"
}
```


Parameter content type: application/json


```
{
  "attachments": [
    {
      "fileContent": "string",
      "fileName": "string"
    }
  ],
  "content": "string",
  "responder": "string",
  "ssn": "string",
  "subject": "string"
```


Response Messages

Figure 20. Create message endpoint with the model containing all the information needed to create a new message

5.2 Reflections

The implementation of this project was not that complex overall, and it was quite easy to get started since almost all the technology and frameworks we used were already in house and well established. We had good guidance from our supervisors and other Crosskey employees. We learned a lot during this project and got a greater understanding of how systems like these are built from the ground up.

We also learned how important communication is when it comes to software development. All included parties have to be on the same page for everything to work properly. Misunderstandings can delay projects and in worst cases cost a lot of money. Fortunately we

managed to complete the project within the given timeframe, despite the communication mishap.

In hindsight, more unit tests could probably have been implemented, but we prioritised getting the implementation in place first and foremost. Another thing to consider is the use of Spring Boot instead of Spring Framework, this would probably help you get started more quickly and easily.

There are already plans for other projects to begin using our REST API, and this is something we had in mind when developing this application. Our application can easily be extended with new functionality and endpoints. It is going to be interesting to see what the future holds regarding our project.

REFERENCES

About Crosskey. (2015, January 12). https://www.crosskey.fi/our-story/

Atlassian. (2021). *Jira*. https://www.atlassian.com/software/jira

Baeldung. (2017, September 12). *Spring's RequestBody and ResponseBody Annotations*. https://www.baeldung.com/spring-request-response-body

Configuration (Spring Framework 5.3.6 API). (2021). https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Configuration.html

Crusoveanu, L. (2016, December 28). *Inversion of Control and Dependency Injection with Spring*. https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring

Data Transfer Object. (2020). https://martinfowler.com/eaaCatalog/dataTransferObject.html

Dependency Injection. (2021). https://www.tutorialsteacher.com/ioc/dependency-injection

EnableSwagger2. (2017, May 20). https://springfox.github.io/springfox/javadoc/2.7.0/springfox/documentation/swagger2/annotations/EnableSwagger2.html

Heller, M. (2020, March 9). *What is Jenkins? The CI server explained*. https://www.infoworld.com/article/3239666/what-is-jenkins-the-ci-server-explained.html

Jenkins. (2021). https://www.jenkins.io/

JSON. (2021). https://www.json.org/json-en.html

Klein Ikkink, H. (2012) *Gradle Effective Implementation Guide,, Pack Publishing Limited*

MIME types (IANA media types). (2021). https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types

MVC - MDN Web Docs Glossary: Definitions of Web-related terms. (2021). https://developer.mozilla.org/en-US/docs/Glossary/MVC

mybatis. (2021). http://mybatis.org

Om Ålandsbanken. (2015, April 20). https://www.alandsbanken.ax/om-oss

Produkt Easit GO. (2019, January 31). https://easit.se/produkter/easit-go/

PropertySource (Spring Framework 5.3.6 API). (2021).
https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/PropertySource.html

Red Hat JBoss Enterprise Application Platform (JBoss EAP). (2021).
https://www.redhat.com/en/technologies/jboss-middleware/application-platform

REST API Documentation Tool. (2021). https://swagger.io/tools/swagger-ui/

Spring Framework Overview. (2021).
https://docs.spring.io/spring-framework/docs/5.1.9.RELEASE/spring-framework-reference/overview.html

Spring makes Java simple. (2021). https://spring.io/

The Official YAML Web Site. (2021). https://yaml.org/

Thompson, J. (2017, September 10). *Using the Spring @RequestMapping Annotation*. DZone.
https://dzone.com/articles/using-the-spring-requestmapping-annotation

Walls, C. (2019) *Spring in Action: Fifth Edition*, *chap. 1.1 What is Spring*, *Manning*

What is Gradle? (2021). https://docs.gradle.org/current/userguide/what_is_gradle.html

What is REST? (2021). https://www.codecademy.com/articles/what-is-rest

What is Swagger. (2021). https://swagger.io/docs/specification/2-0/what-is-swagger/