Thesis, Åland University Of Applied Science, Degree Programme in Information Technology

REAL TIME OPERATION OF NEWSLETTER GENERATION

Martins Animasaun

1:2018

Publishing date: 16.01.2018
Supervisor: Agneta Eriksson-Granskog

EXAMENSARBETE
Högskolan på Åland

Utbildningsprogram:	Informationsteknik
Författare:	Martins Animasaun
Arbetets namn:	Realtids-hantering av nyhetsbrev
Handledare:	Agneta Eriksson-Granskog
Uppdragsgivare:	Paf (Ålands Penningautomatförening)

Abstrakt

I mitt examensarbete skriver jag om hur jag ska förbättra en nyhetsbrev-generator (NLG) som är inte automatiserad men som ska bli automatiserad. Syftet med arbetet är att göra det till en realtids hantering som en runtime-verktyg i realtid hämtar mallen från. Resultatet baseras på följande; kan man utveckla en komponent som kan råda bot på problemen med nuvarande NLG. Följande metoden ska användas: Spring boot application (Java8), Docker container, Maven, Jenkins och JUnit för automattest. Resultatet av det nya NLG upplägget erbjuder mellan komponenten *Email-template-api*. Det förser RESTful API GET och POST till de två komponenterna *Email-sender-api* och *Email-template-gui.*

Nyckelord (sökord)

examensarbete, newsletter generator, spring boot, docker, jenkins, REST API, JUnit, MySQL

Högskolans serienummer:	**ISSN:**	**Språk:**	**Sidantal:**
1:2018	1458-1531	English	35

Inlämningsdatum:	**Presentationsdatum:**	**Datum för godkännande:**
15.12.2017	01.12.2017	20.12.2017

DEGREE THESIS
Åland University of Applied Sciences

Study program:	Information Technology
Author:	Martins Animasaun
Title:	Real-time Operation of Newsletter Generation
Academic Supervisor:	Agneta Eriksson-Granskog
Technical Supervisor:	Paf

Abstract

In my thesis I write about how I will improve an NLG (Newsletter Generator) that is not automated and make it automated.

The purpose of this thesis is to refine the NLG and make it a real time system where a runtime component in real time gets templates from.

The result is based on the following question whether one can develop a service that performs this task automatically and eliminate the identified problems. The following methods are used: Spring Boot application (Java 8), Docker container; Maven, Jenkins and JUnit for automated functional tests.

Result of the new NLG design offers a middle component namely *Email-template-api*. *Email-template-api* provide RESTful API GET and POST to two already existing components *Email-sender-api* and *Email-template-gui*.

Keywords

thesis, newsletter generator, spring boot, docker, jenkins, REST API, JUnit, MySQL

Serial number:	ISSN:	Language:	Number of pages:
1:2018	1458-1531	English	35

Handed in:	Date of presentation:	Approved on:
15.12.2017	01.12.2017	20.12.2018

TABLE OF CONTENTS

1. INTRODUCTION

1.1. Purpose

The purpose of my thesis is to refine Paf's NLG so it becomes a runtime system from which the game system (message-sender) in real time exports templates from. The NLG is a separate offline back-office application in which one manually exports templates to a version control system that further releases the templates into the file system on the production environment. Following are some identified problems with the current NLG: It is error prone and therefore other developer's code are at risk of breaking because one need to copy templates manually (i.e. by hand) from one system to the other where code resides. Each step is daunting; there is no way of testing the templates, and it takes time.

1.2. Method

I will use the programs highlighted below to accomplish my goal

- Spring Boot application (Java 8). More on this in section 3.1.
- Docker container. More on this in section 5.2.
- Maven. More on this in section 5.4.
- Jenkins. More on this in section 5.6.
- JUnit for automated functional tests. More on this in section 6.

It is in the interest of Paf that I use these programs for development, so the new program can easily be ported into their system. Also, I am in contact with my supervisors at Paf during the course of this thesis. Thus, they gave me useful suggestions and ideas. And of course I shall be using the knowledge benefited from both programming and database courses from the university.

1.3. Scope and delimitations

The architecture employed in this thesis comprises of three components, namely: *Email-sender-api*, *Email-template-api* and *Email-template-gui*. They need to work together to get rid of the problems associated with the current NLG. Below is an overview of each component:

- *Email-sender-api* or Message-sender gets email templates from *Email-template-api* and forwards the templates and its contents to Paf's customers that it may concern.
- *Email-template-api* acts as the middleman between *Email-sender-api* and *Email-template-gui*.
- *Email-template-gui* is where a user creates, edits and saves templates onto the disk by a user.

Email-sender-api already exists and is working with the current NLG setup, so it will not be part of this thesis. The, *Email-template-gui* exists too. It is from here templates are copied manually to a version control system and released to a local folder in *Email-sender-api*, therefore, it will not be part of this thesis. Only the *Email-template-api* will be discussed. It is this component that will provide new interfaces to the two already existing components and eliminate the problems with the current NLG setup for good.

2. BACKGROUND

2.1. Employer

This thesis work is made for Paf's site development team (SiteDev). Paf is a Finnish company that operates a legal gambling monopoly on the Ålands Island, Finland. It also has Internet-based gambling and gaming. It operates and maintains casino and gaming activities on a large number of cruise ferries and serves players from countries including Finland, Sweden, Estonia, Spain and Italy (Wikipedia, 2017b).

2.2. Newsletter Generator or NLG

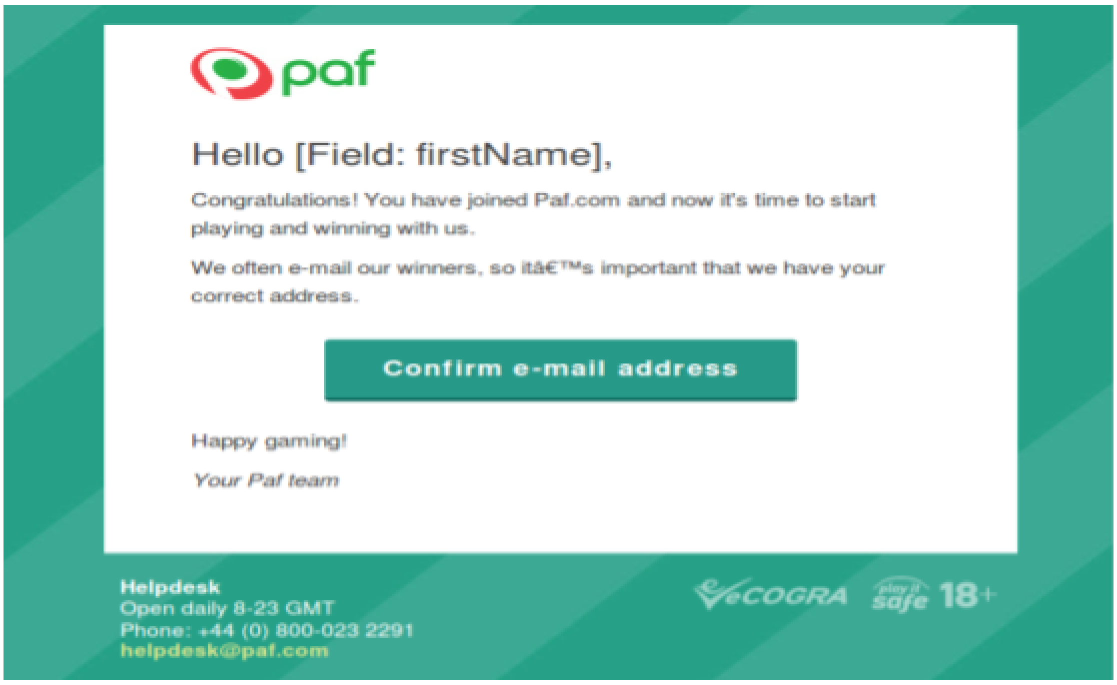
A newsletter is a printed report containing information of the activities of a business or an organization. Newsletters are sent by mail regularly to all members, customers, employees or people that an organization is interested in. Newsletters generally contain one main topic of interest to its recipients (Wikipedia, 2017a). A newsletter generator or NLG is a system one uses to generate and edit newsletters. I have highlighted examples of topics that newsletters which are sent from Paf may contain:

- *Verify your email* (usually when a customer just joined Paf)
- *Password change* (usually when a customer want to reset his/her password)
- *Promo* (usually to regular customers during promotion season)

In essence a topic is what a customer sees as the subject, when he or she get a digital newsletter from Paf.

2.3. Newsletter template

A newsletter template is the body of a newsletter (i.e. the texts that customers can read and so called placeholders) see figure 1. Newsletter templates at Paf are created with *Email-template-gui* and is completely in HyperText Markup Language (HTML).


Hello [Field: firstName],

Congratulations! You have joined Paf.com and now it's time to start playing and winning with us.

We often e-mail our winners, so itâ€™s important that we have your correct address.

Happy gaming!

Your Paf team

Helpdesk
Open daily 8-23 GMT
Phone: +44 (0) 800-023 2291
helpdesk@paf.com

Figure 1. Example of a Newsletter template. [Field: firstaName] is called a placeholder. Email-sender-api fills it with a value. I.e. appropriate customer name.

Table 1 shows attributes with arbitrary values that constitute a newsletter template.

Table 1. Attributes with arbitrary values of a newsletter template

Attribute	***value***
Name	Verify - email
Subject	Please verify your email
Sender_address	info@paf.com
Content	<html> … </html>
Version	1
CreatedDate	2017-11-22T10:39:28Z
Site	Paf.com

Locale	en

Following bullet points summarize each attribute:

- *Name* - use to designate a particular template from other templates
- *Subject* - give customers an overview of the contents of a template or newsletter (non technical perspective)
- *Sender_address* - give the newsletter sender’s address
- *Content* - give detailed information about the subject of a template or newsletter (non technical perspective)
- *Version* - use to differentiate between different versions of a template
- *CreatedDate* - give the timestamp when a template was created
- *Site* - give user’s region.
- *Locale* **-** give user's language

3. EMAIL-TEMPLATE-API

The three components that comprises the new NLG setup is shown in figure 2.

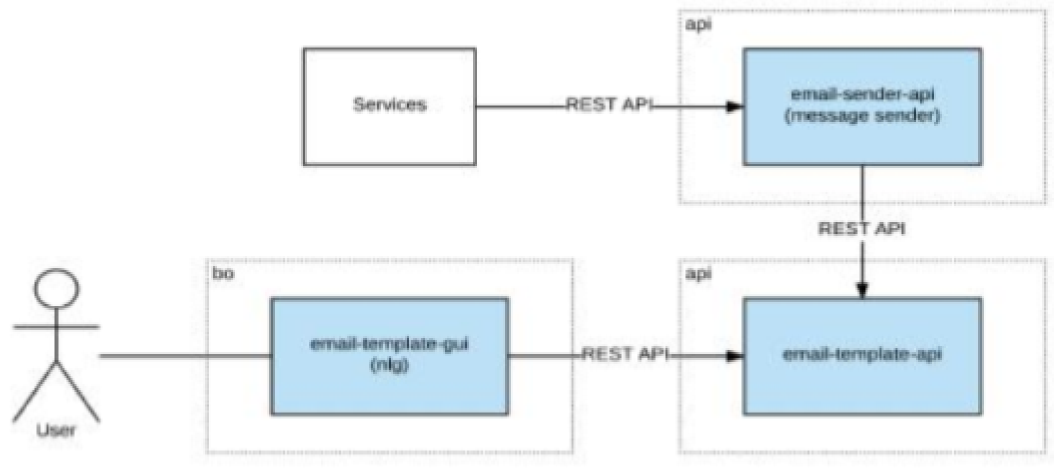


Figure 2. Shows the new NLG setup. From top (colored) is Email-sender-api , Email-template-api *(middle) and* Email-template-gui *(left)*

All three components should work together to create an automated NLG. The middle component is critically examined in this thesis and the next sections.

3.1. REST

Representational state transfer (REST) or RESTful web services are a way of providing interoperability between computer systems on the internet. REST-compliant Web services allow requesting systems to access and manipulate textual representations of Web resources by using a uniform and predefined set of stateless operations. In a RESTful Web service, requests that are made to a resource's Uniform Resource Identifier (URI) will generate a response which may be in XML, HTML, JSON or some other defined format. The response may confirm that some alteration has been made to a stored resource (Wikipedia, 2017c) - using HTTP the kind of operations available include those predefined by the HTTP methods GET, POST, PUT, DELETE and so on. Only GET and POST are used in my thesis.

3.2. Email-Template-Api Restful

Email-template-api is the middle component and the engine which the other two components rely to provide needed functionality. It provides RESTful API (see figure 2), for

Email-sender-api and *Email-template-gui* to use. The two REST API used by *Email-template-api* are explained in the next.

3.3. GET

This method is called by *Email-sender-api*, as its name suggest. What it does is that it returns a specific template when it is called see figure 3.

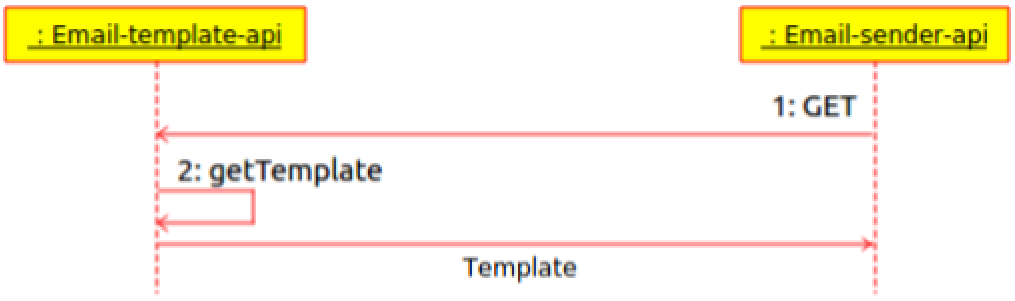


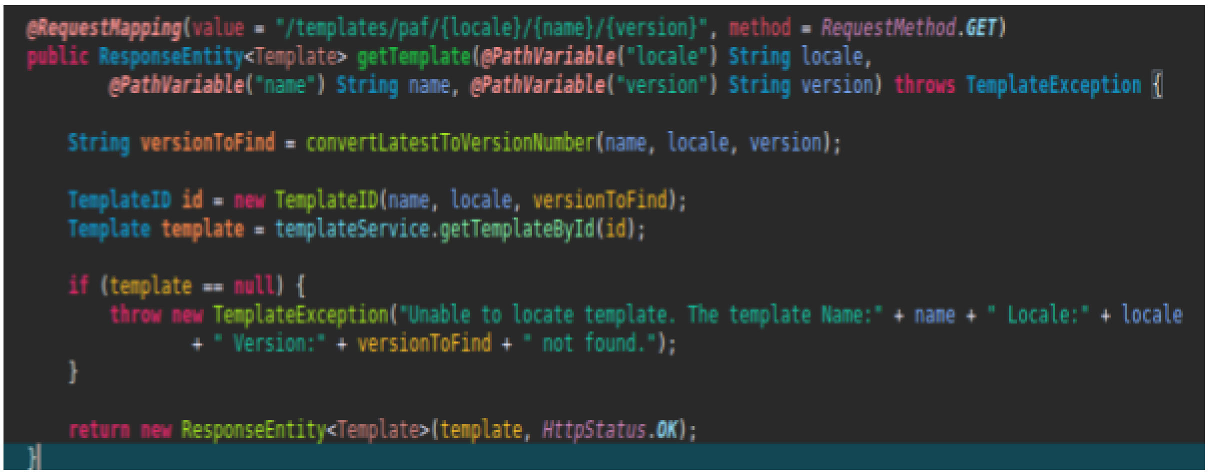
Figure 3. Shows sequence diagram of GET request made by ***Email-sender-api****. getTempate(..) is a method in Maincontroller class. See section 4.3. If a template is found that template will be returned.*

Table 2 illustrates how URI of GET requests sent by *Email-message-sender* is formatted.

Table 2. Illustrates the format of URI of GET requests made by Email-message-sender

Component	***Request***	***URI***
Email-message-sender	GET	*email-template/templates/paf/{locale}/{name}/{version}*

email-template/templates/paf in the URI designates the entry point. The texts in curly braces are placeholders for actual values. Therefore, *locale* should be *sv* - for Swedish customers (i.e customers residing in a region where Swedish is the spoken language); *name* is a placeholder for an actual name of a template e.g. *verify account*, *welcome to paf*, *confirm credit purchase* and so forth, last but not least, is *version* which is a placeholder for a version of a template, thus, *1* for the first version of a template, *2* for the second version, *3* for the third and so forth. To get the latest version of a template, however, one should use *latest*. See figure 4.



```java
@RequestMapping(value = "/templates/paf/{locale}/{name}/{version}", method = RequestMethod.GET)
public ResponseEntity<Template> getTemplate(@PathVariable("locale") String locale,
        @PathVariable("name") String name, @PathVariable("version") String version) throws TemplateException {

    String versionToFind = convertLatestToVersionNumber(name, locale, version);

    TemplateID id = new TemplateID(name, locale, versionToFind);
    Template template = templateService.getTemplateById(id);

    if (template == null) {
        throw new TemplateException("Unable to locate template. The template Name:" + name + " Locale:" + locale
                + " Version:" + versionToFind + " not found.");
    }

    return new ResponseEntity<Template>(template, HttpStatus.OK);
}
```

Figure 4. Code snippet for the GET request mapping. ***convertLatestToVersionNumber(..)*** *converts string* ***latest*** *in the URI to the latest version (in digit) of the requested template.*

I have listed below examples of legitimate URI for making a GET request from *Email-message-sender*:

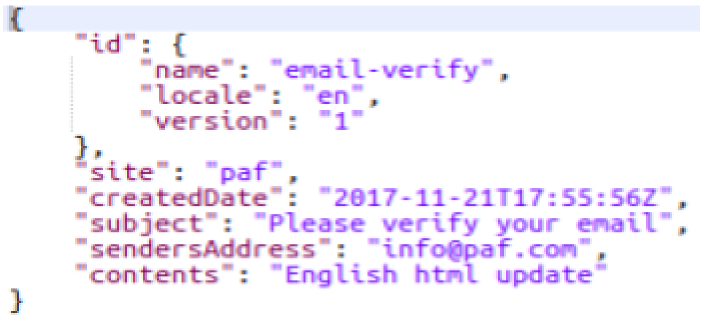
- *email-template/templates/paf/en/verify-email/latest*
- *email-template/templates/paf/sv/welcome/2*

3.4. GET Request Response Body

When a GET request is made to *Email-template-api* the template which is requested has to be identified. Steps to identify the requested template are highlighted and explained below:
As an example, if *Email-sender-api* sends a GET request; GET
email-template/templates/paf/en/email-verify/1
then the following are performed by *getTemplate(..)* see figure 4.

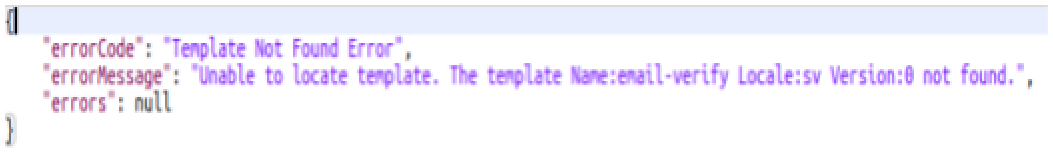
- Extract the version number from the URI and call *convertLatestToVersionNumber(..)*
- Create TemplateID object use values from the URI
- Get a Template object from the persistence service
    - If template exist return HTTP Status Code 200 OK with a response body. See figure 5.

- ○ Otherwise throw an exception. The exception is caught by an exception handler that returns the HTTP Status Code 404 Not Found and an appropriate message. See figure 6.



```
{
    "id": {
        "name": "email-verify",
        "locale": "en",
        "version": "1"
    },
    "site": "paf",
    "createdDate": "2017-11-21T17:55:56Z",
    "subject": "Please verify your email",
    "sendersAddress": "info@paf.com",
    "contents": "English html update"
}
```

Figure 5. JSON document returned from a successful GET request call. HTTP Status Code 200 OK is returned.



```
{
    "errorCode": "Template Not Found Error",
    "errorMessage": "Unable to locate template. The template Name:email-verify Locale:sv Version:0 not found.",
    "errors": null
}
```

Figure 6. Shows a JSON document returned from an unsuccessful GET request call. HTTP Status Code 404 Not Found is returned.

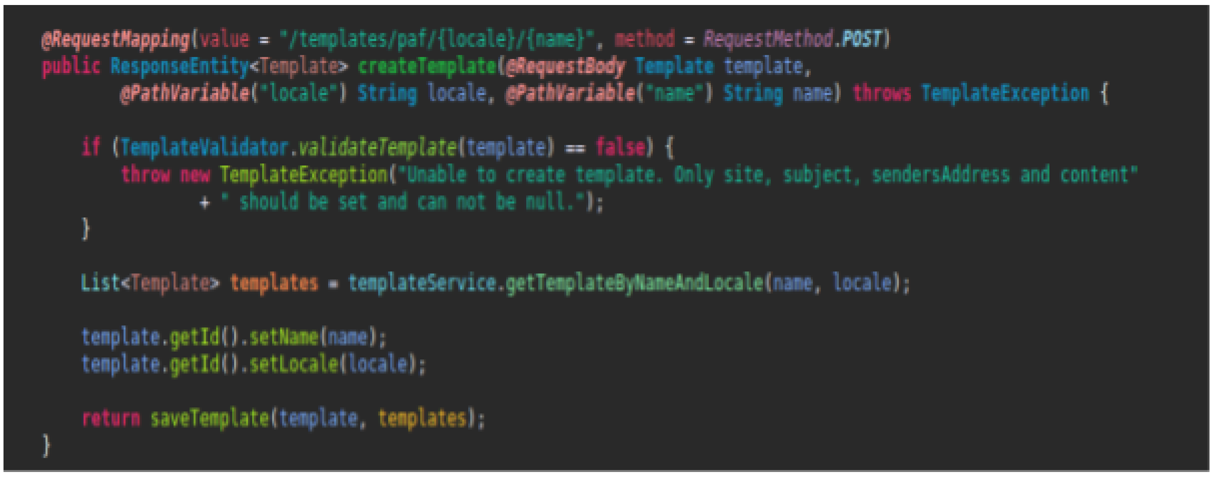
3.5. POST

To save a template *Email-template-gui* need to send a POST request to *Email-template-api*. See Table 3 on how URI of POST requests from *Email-template-gui* should be formatted.

Table 3. Shows URI format of POST request made by Email-template-gui

Component	**Request**	**URI**
Email-template-gui	POST	email-template/templates/paf/{locale}/{name}

email-template/templates/paf designates the entry point see figure 7. The main difference between the POST request URI and the GET request URI is that there are only two placeholders in the POST request URI, namely: *locale* and *subject. locale* is for determining the language of the template contents it is *en* - for English, *sv* - for Swedish and so forth and *name* is for distinguishing a template from another one - it can be anything such as *verify account*, *welcome to paf*, *confirm credit purchase* and so forth.



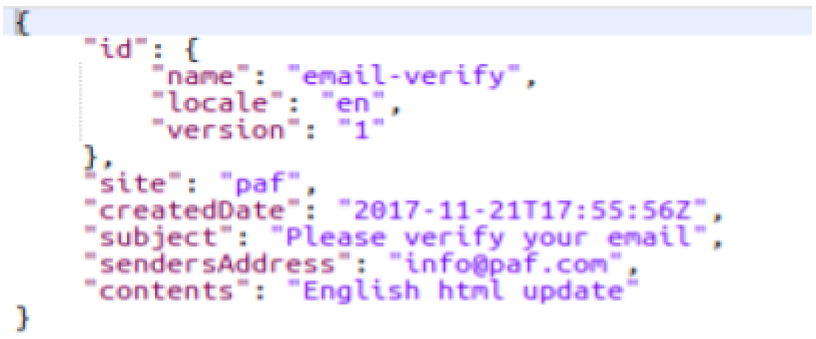
```java
@RequestMapping(value = "/templates/paf/{locale}/{name}", method = RequestMethod.POST)
public ResponseEntity<Template> createTemplate(@RequestBody Template template,
        @PathVariable("locale") String locale, @PathVariable("name") String name) throws TemplateException {

    if (TemplateValidator.validateTemplate(template) == false) {
        throw new TemplateException("Unable to create template. Only site, subject, sendersAddress and content"
                + " should be set and can not be null.");
    }

    List<Template> templates = templateService.getTemplateByNameAndLocale(name, locale);

    template.getId().setName(name);
    template.getId().setLocale(locale);

    return saveTemplate(template, templates);
}
```

Figure 7. Code snippet for the POST request URI mapping. saveTemplate is a method that calls another method saveTemplate from ***TemplateService*** *which extends Spring Boot CRUDRepository interface to persist data.*

3.6. Post Request Response Body

If *Email-template-gui* sends a POST request; POST *email-template/templates/paf/en/email-verify* then the following steps are performed by *createTemplate(..)* in figure 7.

- Find template, if the template does not exist throw an exception is thrown. An exception handler will catch the exception, return HTTP Status Code 404 Not Found and an appropriate message.
- If the template does exist the extracted value from the request body is used to create a template and save the template with *saveTemplate(..)* in the database, and return a response body see figure 8.



```json
{
    "id": {
        "name": "email-verify",
        "locale": "en",
        "version": "1"
    },
    "site": "paf",
    "createdDate": "2017-11-21T17:55:56Z",
    "subject": "Please verify your email",
    "sendersAddress": "info@paf.com",
    "contents": "English html update"
}
```

Figure 8. Response body of a successful POST request by ***Email-template-gui***

3.7. Updating A Template

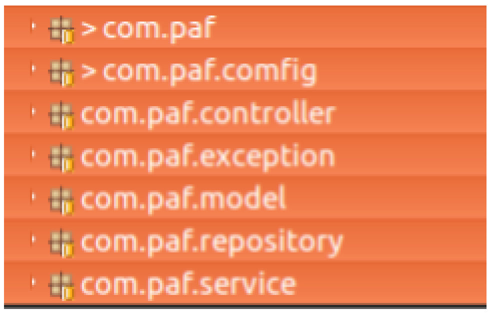
To update a template is a matter of making multiple POST requests such as: POST *email-template/templates/paf/{locale}/{name}* with a request body containing new values for the fields of the template to update in the database. If a template exist with *locale x* and name *y* that template will be updated and persisted in the database. A template with the name y and locale x can be updated multiple times. The update is reflected in the database by incrementing current version numbers of that template in the database. Therefore, a template can have as many version number as necessary. See table 4.

Table 4. Illustrates change in response body when multiple POST request is made on the same URI. Note that version number is incremented.

POST Request	***Template Exist***	***Request Body***	***Response Body***
email-template/templates/paf/sv/welcome	NO	{***"site"***: "paf", ***"subject"***: "Please verify your email", ***"sendersAddress"***: "info@paf.com", ***"contents"***: "html goes here"}	{ ***"id":*** { ***"name":***"welcome", ***"locale"***: "sv", ***"version"***: "**1**" }, ***"site"***: "paf", ***"createdDate"***: "2017-11-22T10:39:28Z", ***"subject"***: "Please verify your email", ***"sendersAddress"***: "info@paf.com", ***"contents"***: "html goes here"}
email-template/templates/paf/sv/welcome	YES	{***"site"***: "paf", ***"subject"***: "Please verify your email", ***"sendersAddress"***: "info@paf.com", ***"contents"***: "html is updated"}	{***"id":*** { ***"name":***"welcome", ***"locale"***: "sv", ***"version"***: "**2**" }, ***"site"***: "paf", ***"createdDate"***: "2017-11-22T10:39:28Z", ***"subject"***: "Please verify your email", ***"sendersAddress"***: "info@paf.com", ***"contents"***: "html is updated"}

4. EMAIL-TEMPLATE-API INFRASTRUCTURE

Email-Template-Api component is divided into seven Java packages see figure 9. The functionality of each of the classes in these packages is explored in the next.



```
> com.paf
> com.paf.comfig
com.paf.controller
com.paf.exception
com.paf.model
com.paf.repository
com.paf.service
```

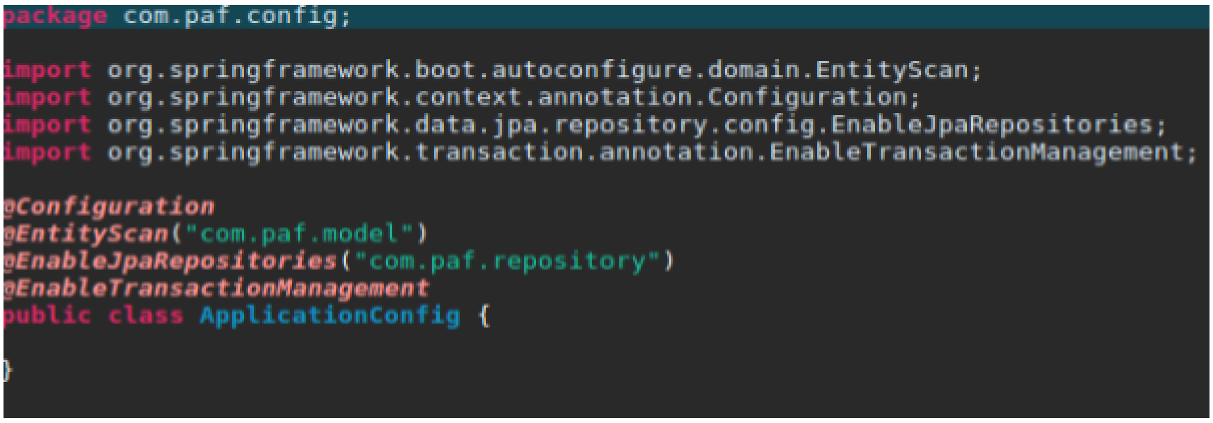
Figure 9. Shows packages of Email-template-api

4.1. Paf

This package contains *EmailTemplateApplication.class*. *EmailTemplateApplication.class* is annotated with *@SpringBootApplication*. It provides a convenient way to bootstrap the Spring application that will be started from the *main()* method.

4.2. Config

This package contains the application configuration class *ApplicationConfig.java*. The class is basically empty but it contains some annotations see figure 10. Its sole purpose is to eliminate configuration problems that occur during testing.



```
package com.paf.config;

import org.springframework.boot.autoconfigure.domain.EntityScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.transaction.annotation.EnableTransactionManagement;

@Configuration
@EntityScan("com.paf.model")
@EnableJpaRepositories("com.paf.repository")
@EnableTransactionManagement
public class ApplicationConfig {

}
```

Figure 10. Snippet from ApplicationConfig.java

When *@WebMvcTest* is used for testing, it looks for a class annotated with *@SpringBootApplication* (see section 4.1) in the same directory, or higher up in the application structure if it doesn't find one. However, if a class that is annotated with *@SpringBootApplication* also has *@EntityScan* or *@EnableJpaRepositories* an error; *at least one JPA metamodel must be present* will occur. This error occurs because the annotations *@EntityScan* and *@EnableJpaRepositories* are together with *@SpringBootApplication* annotation that bootstrap the Spring application but since they are mocked during testing no JPA is actually present. This config class is created to work around this problem and both *@EntityScan* and *@EnableJpaRepositories* are placed there.

4.3. Controller

The controller package contains the MainController *MainController.java*. This class intercepts all request as described in section 3, and use *TemplateService.java* from the package Service to persist data.

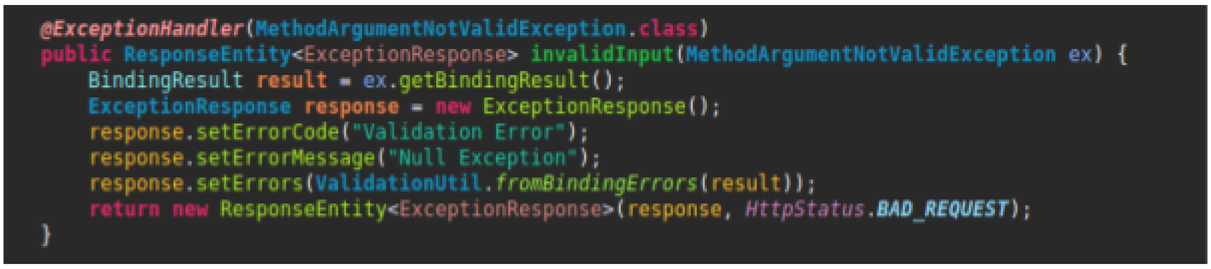
The controller class provides the two methods *getTemplate* and *saveTemplate*. *getTemplate* intercepts GET requests and returns a JSON document that contains a template information if such template exists in the database see figure 7. If no such template exists an exception is thrown. The exception is handled by an exception handler in the package exceptions (see *section 4.4*). *saveTemplate* intercepts POST request (see section 3) and returns a JSON document that contains information about the template that it saved. In a situation where the template already exists the version number of such template is incremented and persisted in the database (see section 3). *saveTemplate* use a pure java method to determine if a template exist and to increment the version number of the template if it exists before it persist the template with new information.

4.4. Exception

This package contains *TemplateException.java*, *TemplateExceptionHandler.java*, and *ExceptionResponse.java* see figure 11. *TemplateException.java* extends java Exception. It is basic and straightforward. It only contains constructor and getter for getting the string that describes the exception.

TemplateExceptionHandler.java handles any exception thrown from *MainController.java* (see section 5.3). The Exception that can be thrown from MainController is an exception of type *TemplateException*. *TemplateExceptionHandler* takes whatever message in the exception object (TemplateException) it receives, and returns it to the client together with an appropriate HTTP status code. In a scenario where *MainController.java* is not able to process a GET request due to invalid URI (i.e. template version or name doesn't exist), and a *TemplateException* is thrown, *templateExceptionHandler(..)* in *TemplateExceptionHandler.java* will be called and the HTTP status code 404 not found will be returned together with an error message.

ExceptionResponse.java is the object that is returned back to the client. It contains an error message and an error code. This class is also basic it only has a constructor and setter and getter methods for the error code and the error message.



```
@ExceptionHandler(MethodArgumentNotValidException.class)
public ResponseEntity<ExceptionResponse> invalidInput(MethodArgumentNotValidException ex) {
    BindingResult result = ex.getBindingResult();
    ExceptionResponse response = new ExceptionResponse();
    response.setErrorCode("Validation Error");
    response.setErrorMessage("Null Exception");
    response.setErrors(ValidationUtil.fromBindingErrors(result));
    return new ResponseEntity<ExceptionResponse>(response, HttpStatus.BAD_REQUEST);
}
```

Figure 11. Snippet from TemplateExceptionHandler.java shows how the classes in the package exception are wired together.

An exception handler keeps the MainController clean and introduces separation of concerns. It also has the advantage of code reusability. An annotation *@ControllerAdvice* is used in *TemplateExceptionHandler.java* to ensure that only the exception message that describes the error that occurred, and a corresponding HTTP status code is returned to the client. Without this annotation, stacks of method calls up to the point where an exception is caught will be included in the response body that the client gets (Chapman Paul, 2013). Since the stack is very long, and the majority of its content is junk to the client anyway, *@ControllerAdvice* is used.

4.5. Model

This package contains *Template.java* and *TemplateID.java*.
Template.java is an entity class annotated with *@Entity*. *@Entity* means that the class is an entity (i.e. it can be persisted), *@Id* is an identifier for identifying a particular entity (ObjectDB Software, 2017). The variables of *Template.java* constitute a template. See table 5.

Table 5. Shows variables and data types of a Template object.

Attribute	***Datatype***
id	*TemplateID*
site	*String*
createdDate	*Date*
subject	*String*
senders_address	*String*
contents	*String*

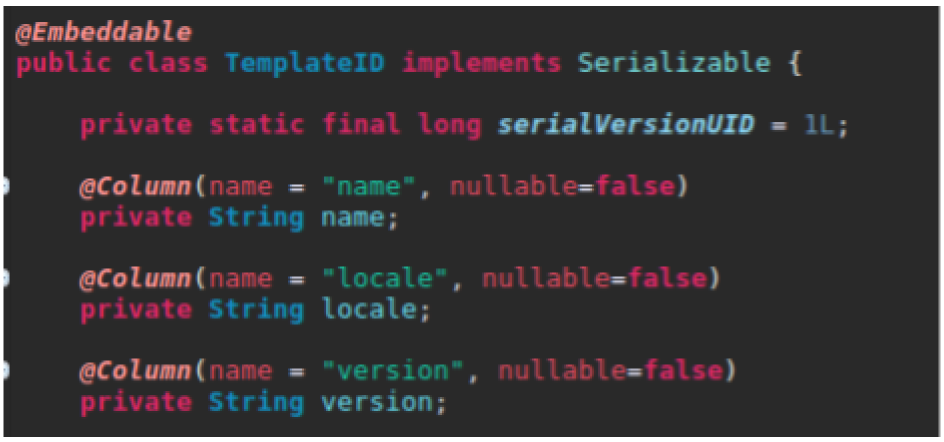
- id: use to uniquely identify a template annotated with *@Id*
- site: *see section 2.3*

- CreatedDate: *see section 2.3*
- Subject**:** *see section 2.3*
- Senders_address**:** *see section 2.3*
- Contents: *see section 2.3*

Template.java also includes some getter and setter methods for setting and retrieving these attributes.

TemplateID.java is an embeddable class. Embeddable classes are user defined persistable classes that function as value types. As with other non entity types, instances of an embeddable class can only be stored in the database as embedded objects, i.e. as part of a containing entity object. Instances of embeddable classes are always embedded in other entity objects and do not require separate space allocation and separate store and retrieval operations (ObjectDB Software, 2017). Embeddable classes do not have an identity (primary key) of their own which means their instances cannot be shared by different entity objects, and they cannot be queried directly. Thus, a decision whether to declare a class as an entity or embeddable requires case by case consideration (ObjectDB Software, 2017).

TemplateID.java contains the attributes for identifying a particular template (see section 6 and figure 11). *TemplateID.java* is used to represent columns in the database that uniquely identify a template since a single column in the database cannot uniquely identify a template. *TemplateID.java* is embedded in *Template.java* with *@Embeddable*.



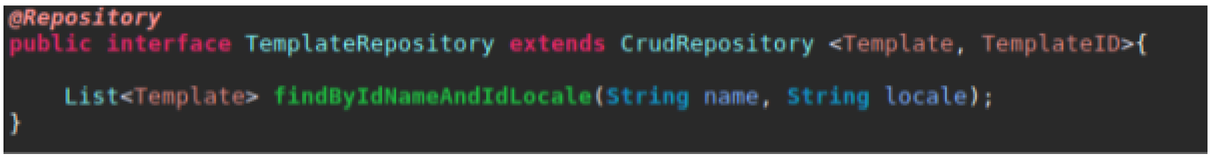
```java
@Embeddable
public class TemplateID implements Serializable {

    private static final long serialVersionUID = 1L;

    @Column(name = "name", nullable=false)
    private String name;

    @Column(name = "locale", nullable=false)
    private String locale;

    @Column(name = "version", nullable=false)
    private String version;
```

Figure 11. Snippet from TemplateID.java shows attributes for uniquely identifying a template in the database.

Other properties of *TemplateID.java* are some setter and getter methods for its attributes. It is necessary that the *TemplateID.java* be serializable and there need to exist implementation for *equals()* and *hashCode()*. The reason for this is it is stated that an entity must be equal to itself across all JPA states: *transient*, *attached*, *detached*, and *removed* (as long as the object is marked to be removed and it still living on the heap) therefore one can't rely on the default Object *equals/hashCode* implementations, since two entities loaded in two different persistence contexts will end up as two different Java objects. Therefore breaking the all-states equality rule and if Hibernate uses the equality to uniquely identify an Object, for its whole lifetime, there need to be right combination of properties satisfying this requirement (Vlad, 2013).

4.6. Repository

This package contains *TemplateRepository.java* which extends Java Persistence API (JPA) CrudRepository for data persistence and retrieval. In addition to *findOne(..)* (JPA) that returns a unique template when given a *TemplateID* Object if the template exist in the database, a custom method interface is supplemented see figure 12.



```
@Repository
public interface TemplateRepository extends CrudRepository <Template, TemplateID>{

    List<Template> findByIdNameAndIdLocale(String name, String locale);
}
```

Figure 12. Snippet from TemplateRepository.java shows a custom method interface.

The custom method returns a list containing templates that have the values of *name* and *locale*. The pattern of the method name *findByIdNameAndIdLocale* is interesting. Since *Template* contains an embeddable (the attribute *id* which is an Object of *TemplateID)* to partially identify a template using some of the attributes of *id*, in this case *name* and *locale* one need to adopt this name pattern. What this name pattern is conveying in other words is this: find templates that have *name* and *locale* in *id* attribute of some templates and return them.

Because *Email-template-api* offers only two major RESTful API, the purpose of *TemplateRepository.java* then is basically to save and retrieve templates from the persistence layer, straightforward and out of the box functionality. Apart from the issue of finding templates by using partial *TemplateID* values everything else is already fixated by the JPA.

4.7. Service

This package contains two Java classes, *TemplateImp.java* and *TemplateService.java*. *TemplateService.java* is an interface that provides two methods *saveTemplate(..)* and *getTemplate(..)*. These two methods are used for saving and retrieving templates and are implemented by *TemplateImp.java*. *TemplateImp.java* use *TemplateRepository.java* (see section 4.5), for persisting and retrieving templates. By using *TemplateService* I am coding to an interface.

5. GUIDELINES

In this section I have summarized guidelines that the project supervisor Paf want me to follow.

5.1. Persistence

The current NLG architecture saves all templates on the disk in the *Email-template-gui* component. For the new system, however, the specification is to persist templates in a database. For this a relational database MySQL is employed. Because JSON data will be saved in the database the design of the database is simple. Coming up with a table for the database was a breeze through - identifying the attribute of the template to be persisted in the database and defining attributes for uniquely identifying a template. Once this is achieved the repository and entity object as explained in chapter 4, can be used to persist and retrieve templates.

Another specification is to use Docker. Therefore, it follows that the database be containerized. See section 6 on how this is achieved.

The property file of the application *application.properties* contains information about the database login credentials and other necessary details for operating with no frills.

5.2. Docker

Docker is built on top of Linux Containers (LXC). Like with any container technology, as far as Docker is concerned, it has its own file system, storage, CPU, RAM, and so on. The key difference between containers and Virtual Machines (VMs which are not discussed here) is that while the hypervisor (VM) abstracts an entire device, containers just abstract the operating system kernel (Vaughan-Nichols, 2017) see figure 13.

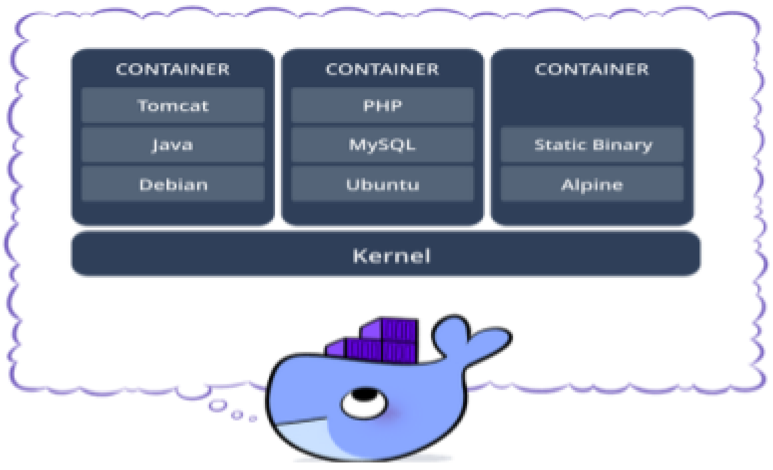


Figure 13. Depict Docker container's architecture (Doc Inc, 2017).

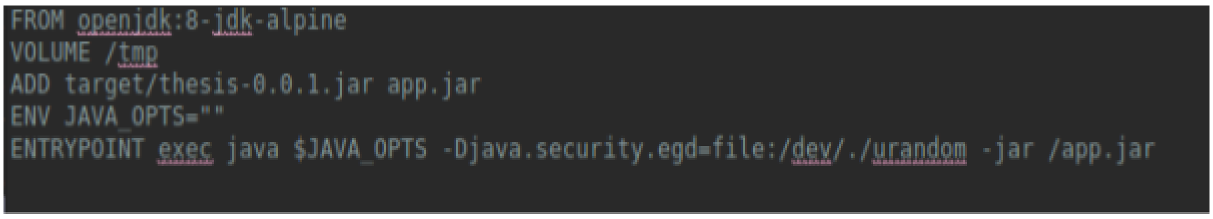
VMs are huge in terms of system requirements. Containers, however, use shared operating systems which means they are much more efficient than hypervisors in system resource terms. Instead of virtualizing hardware, containers rest on top of a single Linux instance as shown in figure 13. This means one can leave behind the useless 99.9 percent of VM junk, leaving behind a small, neat capsule containing applications. However, one thing hypervisors can do that containers can't is to use different operating systems or kernels (Vaughan-Nichols, 2017).

Docker brings several new things to the table that the earlier technologies didn't. The first is that it made containers easier and safer to deploy. Developers can use Docker to pack, ship, and run any application as a lightweight, portable, self-sufficient LXC container that can run virtually anywhere which results in instant application portability and Docker containers are easy to deploy in a cloud. In a nutshell: Docker can get more applications running on the same hardware than other technologies. It makes it easy for developers to quickly create ready-to-run containerized applications and it makes managing and deploying applications much easier (Vaughan-Nichols J. Steven, 2017).

5.3. Email-template-api Dockerfile

A container image is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries and settings. Containerized software will always run the same regardless of the environment since containers isolate software from its surroundings e.g. differences between development and staging environments and help reduce conflicts between teams running different software on the same infrastructure (Doc Inc, 2017). To containerize the *Email-template-api* I had to install Docker on my computer. Once that step was done I then created a Docker Image and a Docker account to push the image.

Docker has Dockerfile that it uses to specify the layers of an image. The Dockerfile for *Email-template-api* is shown in figure 14.



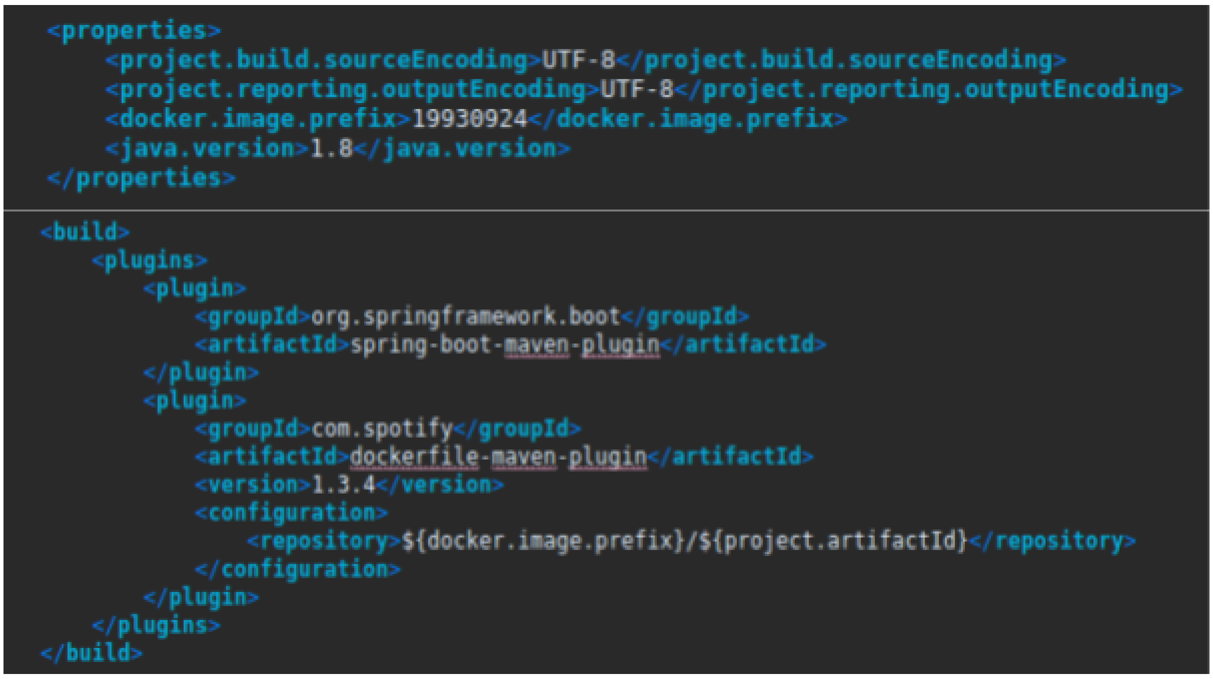
```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ADD target/thesis-0.0.1.jar app.jar
ENV JAVA_OPTS=""
ENTRYPOINT exec java $JAVA_OPTS -Djava.security.egd=file:/dev/./urandom -jar /app.jar
```

Figure 14. Dockerfile for Email-template-api.

The Dockerfile is very simple, that’s all that is required to run a Spring Boot application. The project JAR file is added to the container as *app.jar* and is executed in the *ENTRYPOINT*. *VOLUME* in the Dockerfile points to *"/tmp"* because that is where Spring Boot applications create working directories for Tomcat. Apache Tomcat is a web server and servlet container that is used to serve Java applications by default. The effect is to create a temporary file on my host under "/var/lib/docker" and link it to the container under "/tmp". This helps to reduce Tomcat startup time.

5.4. Email-template-api Docker Image

I use Maven to build the *Email-template-api* service. To build a Docker image with Maven some properties had to be added to the project *pom.xml* (A Project Object Model or *POM* is the fundamental unit of work in Maven. It is an *XML* file that contains information about the project and configuration details used by Maven to build the project) See figure 15.



```xml
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <docker.image.prefix>19930924</docker.image.prefix>
    <java.version>1.8</java.version>
</properties>
```

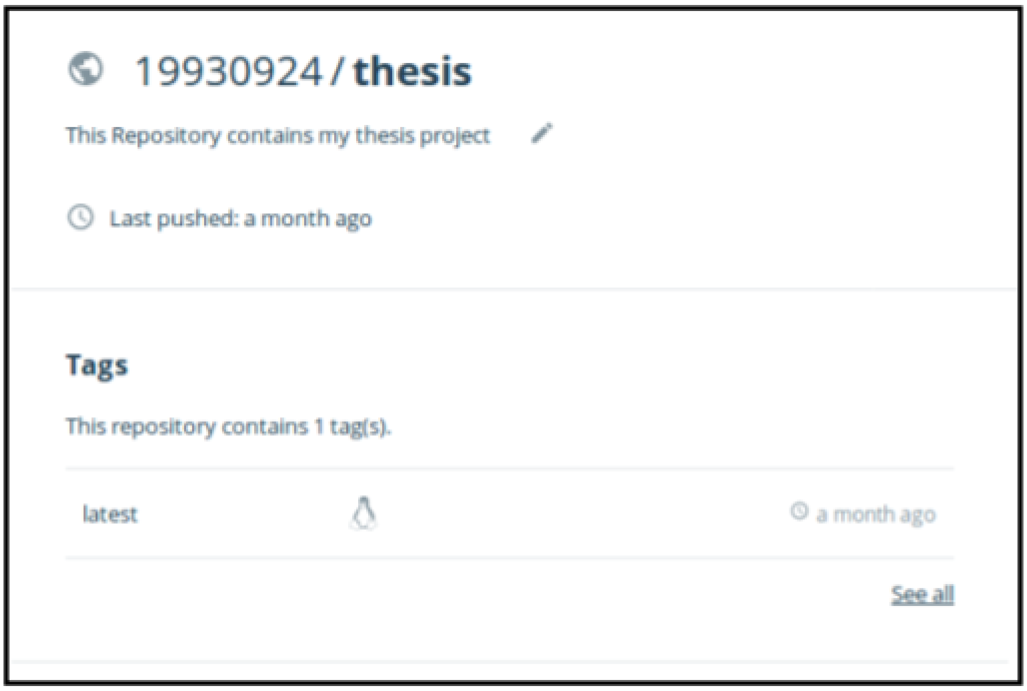
```xml
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
        <plugin>
            <groupId>com.spotify</groupId>
            <artifactId>dockerfile-maven-plugin</artifactId>
            <version>1.3.4</version>
            <configuration>
                <repository>${docker.image.prefix}/${project.artifactId}</repository>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Figure 15. Snippet from pom.xml of Email-template-api

Figure 15 specifies three things:

1. The repository with the image name it will end up here as 19930924/thesis. 19930924 is my username on Docker Hub which happens to be the repository, and image name is thesis (Pivotal, 2017).
2. The name of the jar file, exposing the Maven configuration as a build argument for Docker (Pivotal, 2017).
3. Optionally, the image tag, which ends up as latest if not specified. It can also be set to the artifact id if desired (Pivotal, 2017).

Once completed I used: *./mvnw install dockerfile:build* to build a docker image, and push the Image to my account on Docker Hub with; *./mvnw dockerfile:push* now the image is ready and can be pulled by other developers. See *Figure 16.*


19930924 / **thesis**

This Repository contains my thesis project

Last pushed: a month ago

Tags

This repository contains 1 tag(s).

latest a month ago

See all

Figure 16. Email-template-api Image on my dockerhub account. Latest is the tag, 19930924 is my username and thesis is the repository.

5.5. Containerizing The Persistence Layer

The first step in this procedure, is to download MySQL Server Docker image. To do this I used: *docker pull mysql/mysql-server:tag*. Even though performing this operation as a separate step is not strictly necessary, doing so before creating a Docker container however ensures that my local image is up to date.

Tag in the command is the label for the image version I want to pull (it can be any of the following; 5.5, 5.6, 5.7, 8.0, or latest). If *:tag* is omitted (in my case) latest will be used as label and the image for the latest GA (Generally Available) version of MySQL Community Server will be downloaded.

I used: *docker run --name=email-template-api-db -d mysql/mysql-server:tag* to start a new Docker container for the MySQL Community Server.

--*name* is for supplying a custom name for the server container (*email-template-api-db* in my case), and is optional; if no container name is supplied, a random one will be generated. If the

Docker image of the specified name and tag has not been downloaded by an earlier *docker pull* or *docker run* command. The image is now downloaded after this procedure. Initialization for the container will begin and the container will appear in the list of running containers when running *docker ps*. Once initialization is finished, I checked the random password generated for the root user with: *docker logs email-template-api-db mysql 2>&1 | grep GENERATED*

When the server is ready I ran MySQL Client within the MySQL Server container that is already running and connect it to the MySQL Server. To start MySQL Client inside the Docker container I used: *docker exec -it email-template-api-db mysql -uroot -p*. When asked for password I used the generated password from previous. After I have connected MySQL Client to the MySQL Server I reset the server root password by issuing: *mysql> ALTER USER 'root'@'localhost' IDENTIFIED BY 'newpassword'*

newpassword is substituted with the password of my choice. Once the password is reset the server is ready for use and the image can be pushed to my Docker Hub repository.

5.6. Jenkins

Jenkins is a Continuous Integration (CI) server or tool which is written in Java. It provides Continuous Integration services for software development. It can be started via a command line or a web application server. Continuous Integration (CI) is a development practice that requires developers to integrate code into a shared repository several times a day. It is a process of running tests on a non-developer (i.e. testers) machine automatically when someone pushes new code into the source repository. See figure 17.

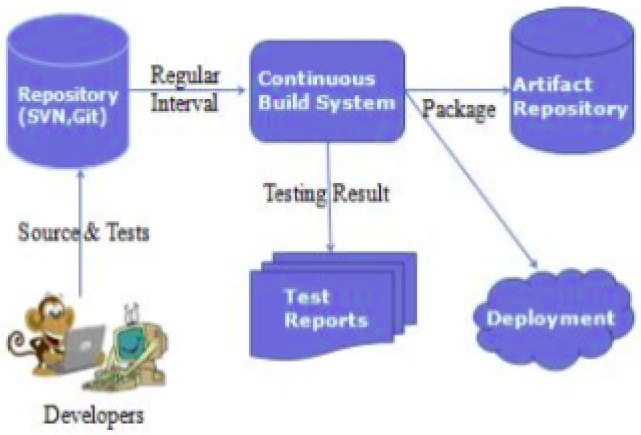


*Figure 17. CI workflow. **Artifact Repository** in CI server will make built snapshot successfully deployed and released to other developers. (Kunja, 2013)*

In such procedure, one can get fast feedback. Fast feedback is very important because one always needs to know right after if a build is broke (i.e. tests failed). In the console, one gets a detailed log messages from which to deduct the reason for job fail (Kunja, 2013).

If jobs are not run occasionally after each push to the repository there will be lots of code changes and it will be difficult to figure out the changes that introduced a problem to a stable repository. However, when tests are set to run automatically on every code push, then one instantaneously know where the cause for the tests failure(s) comes from and who caused it (Kunja, 2013).

Listed below are few reasons why one needs to automate build testing and integration:

1. Developer time is concentrated on work that matters: Most of the work like integration and testing is managed by automated build and testing systems. So the developer's time is saved (Kunja, 2013).
2. Software quality is made better: Issues are detected and resolved almost right away which keeps the software in a state where it can be released at any time safely (Kunja, 2013).
3. Makes development faster: Most of the integration work is automated. Hence integration issues are less. This saves time and money over the lifespan of a project (Kunja, 2013).

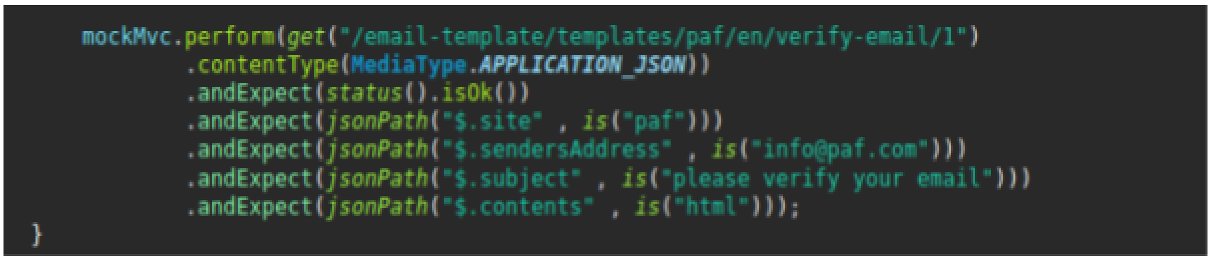
However, I had to test the code myself on the same machine that I used for development. This is mainly because I worked alone. Since setting up a server whose sole purpose will be for testing would require an extra machine and time to save both of these resources. I was forced to use same machine for development and to manually test my code on Jenkins server which I had set up on it. The drawback to this approach of course, is that I have to test the code manually which I quickly got fed up with, so no automation testing whatsoever was employed. On the upside, however, once the code is handed over to my supervisors at Paf it should be easy to integrate to their Jenkins server.

6. TESTING

To test templates the main components of *Email-message-api* need to be tested as well. These main components are *MainController.java*, *TemplateService.java* and *TemplateRepository.java*. Their tests are briefly explained in the next.

6.1. MainController Test

MainControllerTest.java is the test class for the MainController.java (See section 4.3) MainController intercepts all request and use *TemplateService.java* from the package Service to persist data. See figure 18.



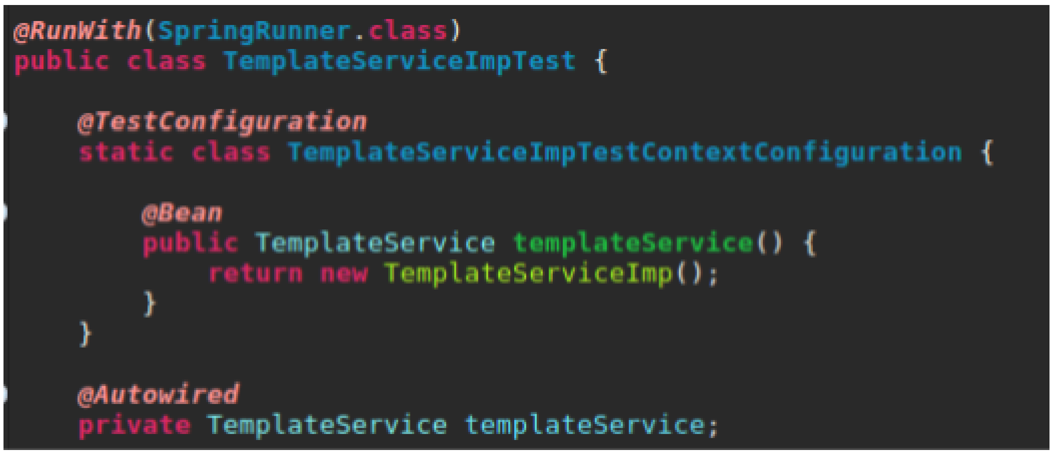
```
    mockMvc.perform(get("/email-template/templates/paf/en/verify-email/1")
            .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.site" , is("paf")))
            .andExpect(jsonPath("$.sendersAddress" , is("info@paf.com")))
            .andExpect(jsonPath("$.subject" , is("please verify your email")))
            .andExpect(jsonPath("$.contents" , is("html")));
}
```

Figure 18. Snippet of GET Request URI test from MainControllerTest.java.

Status code 200 OK will be returned if there is a resource in the URI that was given and the test will pass. If there is no resource in the given URI HTTP Status code NOT FOUND will be returned instead and the test will fail. There are other tests in this class e.g. Tests for URI of a POST request.

6.2. TemplateServiceImp Test

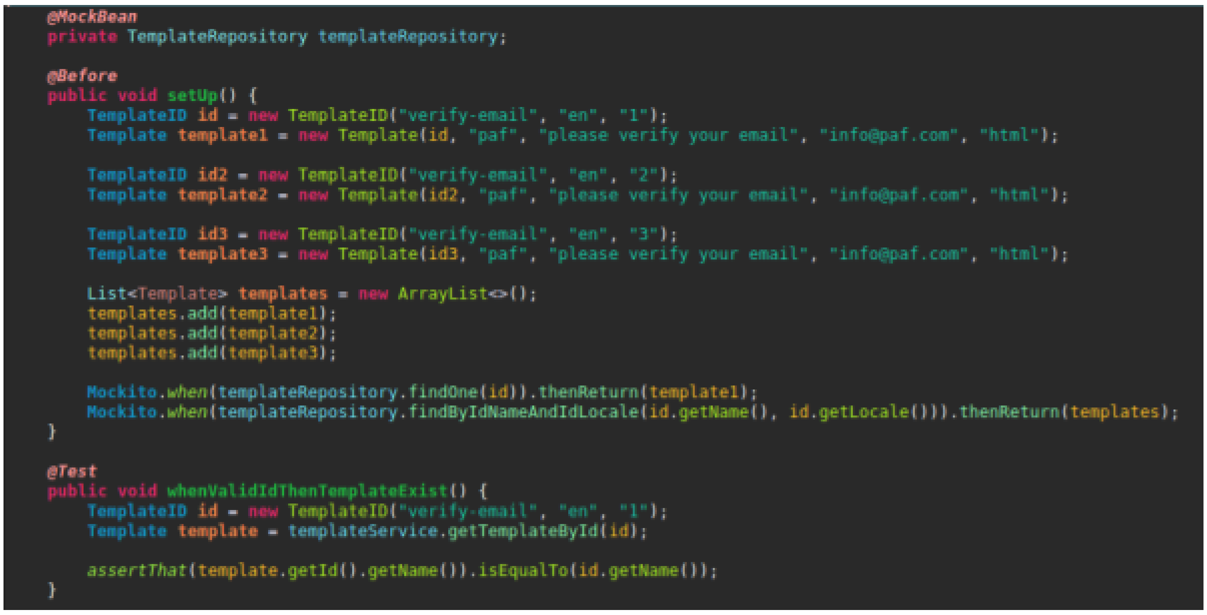
TemplateServiceImpTest.java is the test class for TemplateServiceImp.java (See section 4.6) *TemplateImp.java* implements *TemplateService.java* an interface that provides two methods which are used for saving and retrieving templates. For testing this class, an instance of *Service* class needs to be created and available as a *@Bean*, so it is autowired with *@Autowire* into the test class. This is is achieved by using the *@TestConfiguration* see figure 19.



```java
@RunWith(SpringRunner.class)
public class TemplateServiceImpTest {

    @TestConfiguration
    static class TemplateServiceImpTestContextConfiguration {

        @Bean
        public TemplateService templateService() {
            return new TemplateServiceImp();
        }
    }

    @Autowired
    private TemplateService templateService;
```

Figure 19. Snippet from TemplateServiceImpTest.

@MockBean creates a mock for *TemplateRepository*, which is used to bypass calls to the actual *TemplateRepository* (see figure 20) in the *@Before* annotation a template is created and mockito is used to find and return that template when methods from *TemplateRepository* class are called. Tests are executed in *@Test*. There is another test in this class but omitted for the sake of simplicity.

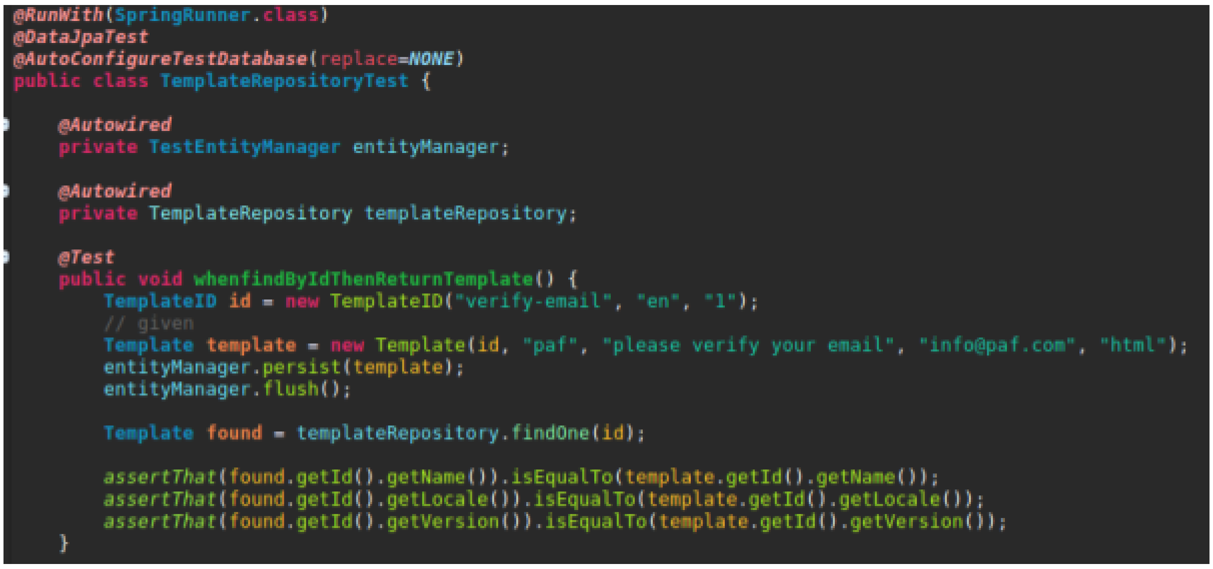


```java
    @MockBean
    private TemplateRepository templateRepository;

    @Before
    public void setUp() {
        TemplateID id = new TemplateID("verify-email", "en", "1");
        Template template1 = new Template(id, "paf", "please verify your email", "info@paf.com", "html");

        TemplateID id2 = new TemplateID("verify-email", "en", "2");
        Template template2 = new Template(id2, "paf", "please verify your email", "info@paf.com", "html");

        TemplateID id3 = new TemplateID("verify-email", "en", "3");
        Template template3 = new Template(id3, "paf", "please verify your email", "info@paf.com", "html");

        List<Template> templates = new ArrayList<>();
        templates.add(template1);
        templates.add(template2);
        templates.add(template3);

        Mockito.when(templateRepository.findOne(id)).thenReturn(template1);
        Mockito.when(templateRepository.findByIdNameAndIdLocale(id.getName(), id.getLocale())).thenReturn(templates);
    }

    @Test
    public void whenValidIdThenTemplateExist() {
        TemplateID id = new TemplateID("verify-email", "en", "1");
        Template template = templateService.getTemplateById(id);

        assertThat(template.getId().getName()).isEqualTo(id.getName());
    }
```

Figure 20. Snippet from TemplateServiceImpTest.

6.3. TemplateRepositoryTest Test

TemplateRepositoryTest.java is the test class for *TemplateRepository.java* (See section 4.6). *TemplateRepository.java* extends Java Persistence API (JPA) CrudRepository for data persistence and retrieval. In this test class just like other test classes discussed thus far, an annotation *@RunWith(SpringRunner.class)* is used to provide a bridge between Spring Boot test features and JUnit. Whenever one is using any Spring Boot testing features in JUnit tests this annotation will be required. Another annotation *@DataJpaTest* provides some standard setup needed for testing the persistence layer which involves the following; configuring H2 (an in-memory database), setting Hibernate, Spring Data and the DataSource, performing an *@EntityScan* and turning on SQL logging. Finally, *@TestEntityManager,* is used for persisting data during testing. *TemplateRepository.java* is autowired with *@Autowired* and used in *assertThat(..)* to verify a template during testing. See figure 21.

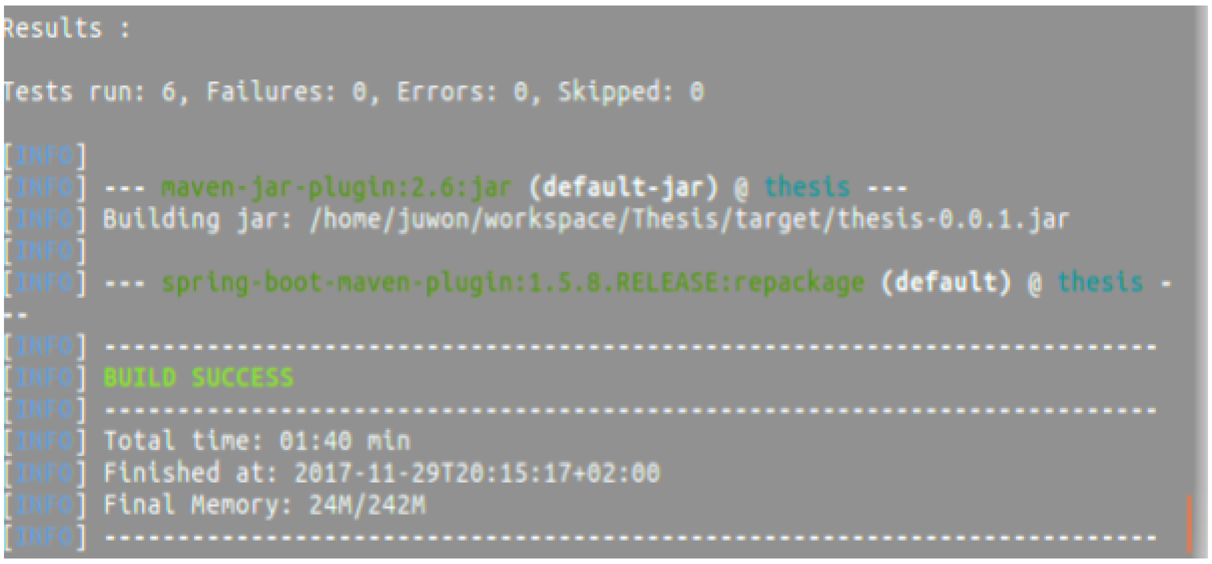


```java
@RunWith(SpringRunner.class)
@DataJpaTest
@AutoConfigureTestDatabase(replace=NONE)
public class TemplateRepositoryTest {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private TemplateRepository templateRepository;

    @Test
    public void whenfindByIdThenReturnTemplate() {
        TemplateID id = new TemplateID("verify-email", "en", "1");
        // given
        Template template = new Template(id, "paf", "please verify your email", "info@paf.com", "html");
        entityManager.persist(template);
        entityManager.flush();

        Template found = templateRepository.findOne(id);

        assertThat(found.getId().getName()).isEqualTo(template.getId().getName());
        assertThat(found.getId().getLocale()).isEqualTo(template.getId().getLocale());
        assertThat(found.getId().getVersion()).isEqualTo(template.getId().getVersion());
    }
```

Figure 21. Snippet from TemplateRepositoryTest.java. There are other tests in the class that have been omitted.

6.4. Test Result

Templates can now be tested for this to be possible some other components of the *Email-template-api* need be tested as well. This components are *MainController*, *TemplateServiceImp* and *TemplateRepostiory*. Figure 22 shows the tests result when I ran the tests on my machine.



```
Results :

Tests run: 6, Failures: 0, Errors: 0, Skipped: 0

[INFO]
[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ thesis ---
[INFO] Building jar: /home/juwon/workspace/Thesis/target/thesis-0.0.1.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:1.5.8.RELEASE:repackage (default) @ thesis -
--
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 01:40 min
[INFO] Finished at: 2017-11-29T20:15:17+02:00
[INFO] Final Memory: 24M/242M
[INFO] ------------------------------------------------------------------------
```

Figure 22. Snippet of the tests result, all 6 test passed, no test failed.

7. CONCLUSION

7.1. Result

The new NLG design offers a middle component namely *Email-template-api*. *Email-template-api* is the focal point of this thesis. It is implemented with *Spring Boot application (Java 8),* and provides RESTful API GET and POST to two already existing components *Email-sender-api* and *Email-template-gui*. These two REST API that *Email-template-api* offers are significant for eliminating any human interaction from the movement of templates or newsletters (customer friendly term) from *Email-template-gui to Email-sender-api* and the customers. The purpose of this component is to help reduce time consumption and money by eliminating the possibility whereby a user mistakenly introduces a bug into program code when manually copying templates from *Email-template-gui*, and making it possible to test templates. To test a template the main components of *Email-template-api* need to be tested using JUnit or Jenkins (for automated testing). Finally, *Email-template-api* needs to be containerized with Docker so other developers can easily run it on their machines with no frills. Each of these point are successfully attained in this thesis.

7.2. Reflection

It is an honour to work with Paf and on this thesis. I have learned a lot during the course of this project mostly about containers Docker in specific, Spring Boot, REST API and Jenkins. I believe that it will be refreshing to know the status of *Email-template-api* and Paf's NLG in the future, maybe by then *Email-template-gui* and *Email-sender-api* will be plugged to the *Email-template-api*.

8. REFERENCES

Chapman Paul. (2013). Exception handling in spring MVC. Retrieved from https://spring.io/blog/2013/11/01/exception-handling-in-spring-mvc

Doc Inc. (2017). What is docker. Retrieved from https://www.docker.com/what-container

Michalcea Vlad. (2013). How to implement equals and HashCode for JPA entities. Retrieved from https://vladmihalcea.com/2013/10/23/hibernate-facts-equals-and-hashcode/

ObjectDB Software. (2017). JPA persistable types. Retrieved from http://www.objectdb.com/java/jpa/entity/types

Pivotal. (2017). Spring boot with docker. Retrieved from https://spring.io/guides/gs/spring-boot-docker/#initial

Shwetha Sneha Kunja. (2013). What is jenkins? . Retrieved from https://vmokshagroup.com/blog/what-is-jenkins/

Vaughan-Nichols J. Steven. (2017). What is docker and why is it so darn popular. Retrieved from http://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/

Wikipedia. (2017a). Newsletter. Retrieved from https://en.wikipedia.org/wiki/Newsletter

Wikipedia. (2017b). Paf. Retrieved from https://en.wikipedia.org/wiki/Paf_(company)

Wikipedia. (2017c). Representational state transfer. Retrieved from https://en.wikipedia.org/wiki/Representational_state_transfer