jamk.fi

Spamler

Email traffic generator

Pavel Kutáč

Bachelor’s thesis
May 2016
School of technology
Bachelor's Degree Program in Information and Communications Technology

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Description

Author(s)	Type of publication	Date
Kutáč, Pavel	Bachelor's thesis	May 2016
		Language of publication: English
	Number of pages: 56	Permission for web publication: yes

Title of publication

Spamler

Email traffic generator

Degree programme

Information and Communications Technology

Supervisor(s)

Salmikangas Esa

Assigned by

Silokunnas Marko

Abstract

The main purpose of this thesis is to describe the whole development process of the Spamler application. The main goal of the application is to generate email traffic for training and testing tasks in JYVSECTEC company. Spamler should send emails for multiple recipients with HTML or plain text template. The HTML template can also contain embedded images, which must be inserted into emails as attachment.

Described is the whole process from the beginning, the collecting and writing of the requirements through designing the database and then creating the main API, and the thesis finishes with the explanation of the created web-based client.

The backend of the service was developed in Go programming language designed by Google, as REST API. All emails are sent in goroutines, which is equivalent to threads in other programming languages. The persistent storage is implemented with the SQLite database system and the configuration of the application is saved in the configuration file in the TOML format.

Users can manipulate the service only via this API using JSON format for data transfers. For better user experience, a client is created as web page in HTML, CSS and JavaScript communication with the created API.

Keywords/tags (subjects)

Go, email traffic, spam, REST API, service, web-based

Miscellaneous

Contents

Figures

Abbreviations

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CPU	Central processing unit
CSS	Cascading styles sheets
CSV	Comma-separated values
DBMS	Database management system
DOM	Document Object Model
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
I/O	Input/output
JSON	JavaScript Object Notation
MVC	Model-View-Controller
OOP	Object-oriented language
REST	Representational State Transfer
RGCE	Realistic Global Cyber Environment
SGML	Standard Generalized Markup Language
SMTP	Simple Mail Transfer Protocol
TLS	Transport Layer Security
TOML	Tom's Obvious, Minimal Language
URL	Uniform Resource Locator
XML	Extensible Markup Language

1 Introduction

1.1 Internet security

The internet is a worldwide network used by more and more devices every day. One can connect to the internet via a mobile phone almost everywhere. Also, cars and household appliances as a fridge, coffee maker or washing machine will be possible to connect to the internet very soon. With the increasing network also security and privacy are hot topics nowadays because many people put their personal data into clouds and remote machines. JYVSECTEC project develops, researches and trains in cyber security, which is more important every day.

Even though there are many messengers and communication applications e.g. Facebook, WhatsApp, Skype and many others, email is still the most used platform, and also most spam messages come by electronic mails. Those undesired messages are also very dangerous and great spam filters are needed for preventing users against those threats.

The task was to create an application for generating email traffic, which will help during training and testing by sending emails automatically. Users of Spamler can produce much bigger traffic by this application than sending messages separately one by one. Also, there will not be needs for people sending emails, and those people can be used somewhere else.

1.2 JYVSECTEC – Jyväskylä Security Technology

JYVSECTEC at JAMK University of Applied Sciences, in the Institute of Information Technology was started as a project in September 2011 to face a topical challenge. The aim of the project has been to create in Central Finland one of Finland's leading research and development and training centres and develop the national and international co-operation network of companies and actors. With the RGCE development environment (Real Global Cyber Environment) JYVSECTEC has been able to offer modern facilities as well as expertise and systems to develop the cyber

security knowledge of organi-sations and create systems to implement cyber security exercises. (About JYVSECTEC, 2016)

1.3 RGCE - Cyber Operation Environment

The environment models dozens of worldwide ISPs (Internet Service Provider) of various sizes, public IP addresses and their real geographical locations. The modelled Internet service providers provide Internet core services e.g. realistic Domain Name Service (DNS), Network Time Protocol (NTP) and WEB services (e.g. news sites, cloud services, social media). In addition, Internet service providers have also customer networks that are used to produce automated user traffic thus modelling the consumer connections of the service providers. (Cyber environment, 2016)

The environment has been implemented using modern cloud service architecture combined with virtualization and flexible connectivity of physical devices. This so-called hybrid model offers a cost-effective, versatile and flexible way to build e.g. a company's operation environment with its services and operation processes. The built environment not only has to model one outlet/location but the environment can model various topologies. RGCE unites physical devices (e.g. routers, switches or IT security devices) with the virtualization resource (operating systems, information security products etc.) (Cyber environment, 2016)

The users of the virtualization resource are offered so called virtual Data Center environment where the users either have ready implemented environments or they can themselves build virtual machines and use them safely via a Web browser without endangering contamination of their own device or being connected to the environment directly. (Cyber environment, 2016)

2 Requirements

2.1 Why to collect the requirements first

If there is some development task, not just in an IT area, analysis and collecting requirements are starting points. This is a very important part however, many times omitted. As mentioned in many tutorials and books, if there is some inconsistence in requirements, price and effort for fixing or remaking will be with passed time bigger.

The thesis was started with collection of basic requirements about an application called Spamler which is an application for generating email traffic to multiple recipients. Users can control execution and settings in web GUI. The main units are campaigns, where the user can specify a list of recipients' emails and a template for emails. Spamler also supports embedded image attachments and tracks the history of email traffic.

In the following chapters are described the functional requirements, non-functional requirements and also constraints of the application. Functional requirements describe what application shall do in specific actions. However, non-functional requirements, also called quality factors or quality attributes, describe what the system should accomplish.

2.2 Campaigns

Campaign is the main execution unit of Spamler. At the same time, Spamler can execute multiple campaigns, however, maximum one instance of each. Campaigns can be executed repeatedly, automatically or manually.

For each campaign the user can set:

- a name of a campaign,
- a list of recipients,
- a list of possible senders,
- a template for emails,
- delays in seconds between emails and continuous executions.

The user also can see a history of all execution and can control the campaign flow by starting, pausing and stopping the process.

Valid campaign settings

The campaign is valid if at least one valid email address exists in the recipient's list and at least one in the sender's list, a template is selected and the name of campaign is not empty. If those data are not valid, the campaign cannot be started.

2.2.1 Campaign's instance execution

If all campaign's data are specified and valid, Spamler can start with sending emails to recipients in the campaign. The validity of the campaign should be checked before each start, because some of the recipients or senders could be removed, when campaign is stopped.

The execution starts when a user presses the Start button, this will the change state of the campaign from Stopped to Running. Spamler will start sending emails to all recipients in the list and waits for set delay before sending an email to the next recipient. The sender of the email is randomly picked from the sender list for each email, however, sender and recipient cannot be the same. If there is only one sender, and the same email is in the recipient list, the program will log an error and ignores that email in the recipients list.

During the execution the user can stop the campaign process, which stops sending emails, and change the state to Stopped. If the user runs the same campaign again, the program starts sending emails from the beginning of the list.

The user also can pause execution, which will change state to Paused. This will not destroy the campaign instance, so a new instance cannot be run during the paused state. If the execution is paused during waiting, the remaining time does not need to be stored. After resuming it can continue immediately. When the campaign is resumed from the paused state by the user, the state is changed back to Running. Spamler then continues sending emails to the next recipient in the list, and it does not start from the beginning. The campaign can be stopped in paused state as well.

When Spamler sends emails to all recipients in the list and the campaign is not continuous, the campaign is automatically stopped. All possible campaign states are:

- running,
- paused,
- stopped.

2.2.2 Continuous campaign

If the campaign is continuous, the delay in seconds must be specified, however, can be 0. When the end of the list is reached, Spamler waits for the given delay before it starts sending emails again from the beginning of the recipients list.

During the waiting time, the state of the campaign is still running. If the campaign is paused during waiting time, the remaining time is not tracked. When Spamler is resumed from the paused state, it starts immediately sending emails from the beginning of the list.

2.2.3 History of campaign execution

Spamler will track the history of all executions. The execution starts, when the campaign is started and ends, when it is stopped. Reaching the end of the list and starting sending from the beginning in a continuous campaign is still considered as one campaign execution. Reaching the end of the list for non-continuous campaigns means the end of the execution.

For each execution Spamler keeps tracks on how many emails were sent successfully and how many errors occurred. The start time of the campaign execution and end time are tracked as well. The start time is set when the state of the campaign is changed from Stopped to Running. The end time is saved when the state is changed from Running to Stopped. Pausing and resuming will not affect start or end time.

All occurred errors are tracked with the recipient to whom the email was sent, time and an error response from a server. The history of campaign execution and errors are read only, and once they are saved, they cannot be modified or deleted. The only possibility to remove history is to remove the whole campaign from the system.

2.2.4 List of recipients in campaign

The campaign consists of the list of recipients' emails, which is separated from other campaigns and does not affect others lists. Each recipient consists of a name, surname and email, which must be unique. Email cannot be inserted into the same list twice and must contains valid email address, first name and last name cannot be empty.

Multiple recipients can be added by parsing CSV file in client. Lines starting with # sign are ignored and the first non-ignored line is used as a line with column names. CSV must contains at least 3 columns:

- first_name,
- last_name,
- email.

Those data are parsed and stored, others are omitted.

All data about recipients can be used as placeholders in templates. The list of recipients cannot be changed if the campaign state is different from stopped.

2.2.5 List of senders in campaign

The sender list has the same restrictions as the recipients list and must consist of a name, surname and email as well. The email must be a unique valid email address, and name and surname cannot be empty.

Importing from CSV file is also possible to sender's list, and if the campaign is not stopped, the list cannot be changed. The same email can be inserted into senders list and recipients list, however, a user must be sure that for each recipient is at least one sender with different email address.

2.2.6 Campaign page

In a campaign page users can create a new campaign, modify all information in old campaigns, or delete whole campaign. Deleting a campaign will also erase the whole execution history with the occured errors. It should not be forgotten that updating or

deleting is possible only in Stopped state. For this purpose, the state must be visible for user.

The campaign consists of a campaign name, list of recipients, list of senders, select box for selecting a template and link to the history of executions. Also, buttons to Start, Pause or Stop campaign execution must be visible, depending on the current state.

2.3 Templates

2.3.1 Template structure

Users can create own plain text or HTML templates. Those templates are used to generate email in campaigns, and placeholders are replaced by data from recipients list. Also, image attachments can be embedded into HTML templates.

For each template, user can specify:

- a name of template,
- a subject of emails,
- a type of template, if is plain text or HTML,
- a content of template,
- attachments.

Updating, deleting and creating new templates must be possible, however, no modifications can be proceeded, if the template is used by a non-stopped campaign. If thetemplate is used by some campaign in any state, deletion cannot be done at all.

Valid template settings

Template is valid if a name and subject is specified, plain text or HTML type is chosen and content of template is not empty. Otherwise template cannot be saved.

Templates list page

In a list of templates users can see all templates, name, subject, type of template, status if template is editable, and there is also a quick action button to see details or remove the template, if is not used.

There must be also Create new template button.

2.3.2 Template page

On a template page users can modify all information in old templates, delete the whole template or just see the information about the template. A template can be saved if all data are valid. If at least one running campaign uses a template, that one cannot be edited. Even if all campaigns using the template are stopped, that template still cannot be deleted.

The template content will be edited in simple textarea for plain text and HTML as well. Under editing field there will be help, how to put placeholders or attachments into template.

Users can render a template without saving it, to see if the template is working correctly. For plain text only the placeholders will be replaced, for HTML also image attachments will be inserted and Sampler will show the final rendered template, not as text with HTML tags anymore.

2.3.3 Attaching images into template

With attachments, users can attach images into the template. For every attached image, Spamler will show placeholder, which users should use for putting image into the template content. Those placeholders will be replaced by an image if they are attached. When the users remove an attachment from template but keep a placeholder in the template content, that placeholder will stay in the message as simple text.

If the user wants to use an attachment, which does not exist, he should be able to upload the considered image on the template editing page without leaving.

2.4 Attachments

Users can upload images as attachments, which can be later used as parts of templates. An attachment can only be an image in JPEG or PNG format. Multiple attachments can be attached to one template and each attachment can be used in multiple templates. The maximum size per each image cannot exceed 5MB.

Attachments list

Users can see a list of all uploaded attachments, open them, change the name of the attachment and also delete them if they are not used in any templates. Users can upload one or multiple images at once.

Name of attachment

Each attachment can be renamed before sending by an email. If users upload an attachment, they can set a different name, which will be later used by templates and attachments.

2.5 Non-functional requirements and constraints

All the requirements described in the previous chapters are the functional requirements and a non-functional requirements with constraints are written in the following chapter. Constraints specify some restrictions about programming languages and used libraries due to used systems, which cannot be changed.

Concurrently running campaigns

All campaigns will run concurrently by goroutines and will not have any side effect onto others. This approach is selected for more smooth process, when waiting for an answer from SMTP server or a database. It will not stop the execution of the whole application and more tasks can be done at the same time. The campaign can be stopped, paused or run without any communication to others campaigns.

Persistent storing data and state

All data filled by user, as well as states of campaigns and already proceeded recipients, must be persistently saved for unexpected crashes or an accidentally killed program. After the program is restarted and campaigns are resumed back, the program must continue sending from last recipient further, as if there had been no crash.

Graphical user interface and configuration

Managing campaigns, templates, attachments and all control of Spamler program will be done via GUI (graphical user interface). Users will have to configure just basic settings for Spamler program only before the first running. Later changes of configuration are needed only in specific cases, as changing the connection to persistent storage or SMTP server. All basic actions described in the requirements are done by web GUI.

Authentication and security

The client will use HTTP Basic authentication to authenticate the user's login and password. A list of available login-password pair is specified in the configuration file. There is no possibility to do registration, send forgotten password to email, or another password recovery.

The application will use secure communication between client and server using HTTPS protocol with TLS. If users will try to access system on a non-secure URL, they will be redirected to the secure URL immediately.

There is no need for authorization and all logged users will have same access rights and possibilities in system.

Constraints

Sometimes it is better to collect functional requirements first, and then decide what will be the best option, however, in this case most of the systems had already been decided and set as constraints and they were followed. More detailed information about each technology is available in Chapter 3.

The backend part of the application must works as server side REST API written in Go language and run as Linux service. All errors or warnings are sent to syslog, the facility and log level of syslog is configured by a file. The configuration is made using TOML (Tom's Obvious, Minimal Language).

The frontend is made as a web application, which will use prepared REST API to communicate with the server. The user interface should be implemented with HTML, CSS and JavaScript, however, more concrete techniques were not specified.

The Whole final program should be distributed by RPM and DEB package for an easier installation on Linux servers.

3 Technologies

3.1 Go language

The Go programming language is an open source project to make programmers more productive. Go is expressive, concise, clean, and efficient. Its concurrency mechanisms make it easy to write programs that get the most out of multicore and networked machines, while its novel type system enables flexible and modular program construction. Go compiles quickly to machine code yet has the convenience of garbage collection and the power of run-time reflection. It's a fast, statically typed, compiled language that feels like a dynamically typed, interpreted language. (Go programming language, 2016)

Go has somewhat different syntax, than programmers can be used to from languages like C, C++, Java but also Python. The major difference is in basic variable declaration, when data type is after variable name, not before. Go is not an object oriented programming language, therefore, the programmer cannot define classes, inheritance, override methods and others techniques of polymorphism, however, a function could be declared as method of a struct, which gives to the developer similar approach as classes and methods in OOP languages.

Primitive data types are the same as in others languages containing int, string, float, byte. The numeric types could be more specified, by range for 32-bit or 64-bit. User can use also composite data types as structs, arrays, slices and maps.

Array is defined with its size and cannot be resized, it is the same as array in C or C++. Slices are more like vectors in C++ or Lists in Java and C#. Size is not fixed and the programmer can add or delete items without reallocating the whole array. Working with slices is very similar to Python, where the developer can specify range for accessing part of a slice. Maps are similar to HashMaps in Java, where index could be also string as another data type.

Even though the programmer cannot allocate or deallocate memory directly, pointers exist in Go language and they are used all the time. There is no need to care about memory management because the garbage collector is used. With that combination users get a very fast program with almost no probability of memory leaks.

Code in Go is separated into packages, which the developer can import when needed. In basic installation there is a huge amount of packages for basic use with strings or conversions, but also more complex use like HTTP server and others network operations. If a package is not present in basic installation, it can easily be downloaded by specifying their URL address, mostly to GitHub. Following code shows syntax of Go language, however, it will not be able build it, because Go is strict. Variable “a” and “b” are not used, which is evaluated as error. The same rule is applied also to unused imports.

```
import "fmt"

// Example of new variable a (explicit type) and b (implicit type)
var a int
b := "This variable will be string"
// Definition of new data type, which is struct
type Person struct {
    Name  string
    Age   int
    IsMan bool
}
// Function without parameters, returning string.
// This function belongs to Person struct defined by (p *Person)
func (p *Person) ReturnAsSentence() string {
    return fmt.Sprintf("Person %v is %d years old", p.Name, p.Age)
}
// Create new variable as pointer to instance of Person struct
person := &Person{"Pavel", 23, true}
fmt.Println(person.ReturnAsSentence())
```

3.2 SQLite

SQLite is an in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. The code for SQLite is in the public domain and is thus free for use for any purpose, commercial or private. SQLite is the most widely deployed database in the world with more applications than we can count, including several high-profile projects. (About SQLite, 2016)

SQLite is an embedded SQL database engine. Unlike most other SQL databases, SQLite does not have a separate server process. SQLite reads and writes directly to ordinary disk files. A complete SQL database with multiple tables, indices, triggers, and views, is contained in a single disk file. The database file format is cross-platform - you can freely copy a database between 32-bit and 64-bit systems or between big-endian and little-endian architectures. These features make SQLite a popular choice as an Application File Format. Think of SQLite not as a replacement for Oracle but as a replacement for fopen(). (About SQLite, 2016)

From the beginning some database had to be used for persistent saving data and there was decision between two technologies, MySQL and SQLite. Each of them has some pros and cons. SQLite was chosen because its limitations are not so significant to limit the project needs. The second reason was the much easier deployment and distribution. SQLite is not a standalone server, however, whole the library is included directly in the application. The administrator does not need to install another tool on target computer, just simply install the application, which contains everything important.

Basic limitations faced was the lack of data type for saving date and time values and ENUM data type, however, for date and time the programmer can use integer and save time as UNIX Timestamp, or save it as string in ISO8601 format. There is also a third possible way, to save it as real number of days since 24.11.4714 B.C. With all of those interpretation it is possible to use built-in date-time functions.

There is no fully functional alternative for ENUM data type. Database designer has to use basic integer, or text. Integer was used here for saving those values, however, there is no constraint check if that value is in the range or one of the possible values.

Data types in SQLite

Most SQL database engines (every SQL database engine other than SQLite, as far as we know) uses static, rigid typing. With static typing, the datatype of a value is determined by its container - the particular column in which the value is stored. (Datatypes in SQLite Version 3, 2016)

SQLite uses a more general dynamic type system. In SQLite, the datatype of a value is associated with the value itself, not with its container. The dynamic type system of SQLite is backwards compatible with the more common static type systems of other database engines in the sense that SQL statements that work on statically typed databases should work the same way in SQLite. However, the dynamic typing in SQLite allows it to do things which are not possible in traditional rigidly typed databases. (Datatypes in SQLite Version 3, 2016)

SQLite contains only the following data types:

- NULL,
- INTEGER,
- REAL,
- TEXT,
- BLOB.

Authors are also applying a method called affinity, which means that datatypes of columns are not strict, but more dynamic like in dynamic typed languages. Even if a column is specified as type INTEGER, the string could be saved there as well. This behavior could be useful with some dynamic typed languages like Python or PHP, however, could be very harmful with static typed languages like Go, which were used here. A scenario could be imagined, when somebody will update database not by their program and some value of different type will get into database. In SQLite no errors will occur, however, it can be a reason of program crash. There does not have to be an implicit cast between the type used in program and the type returned by database.

3.3 Basic access authentication

One of the requirements was also the restricted access into API and application. The decision was made for one of the easiest ways, Basic access authentication, sometimes called HTTP Basic authentication.

If the user is coming to the web page for the first time, the authentication header in the request will not be present. Because of this, the server returns HTTP code 401 Authorization Required. The user's browser will react and show login form as illustrated in Figure 1.

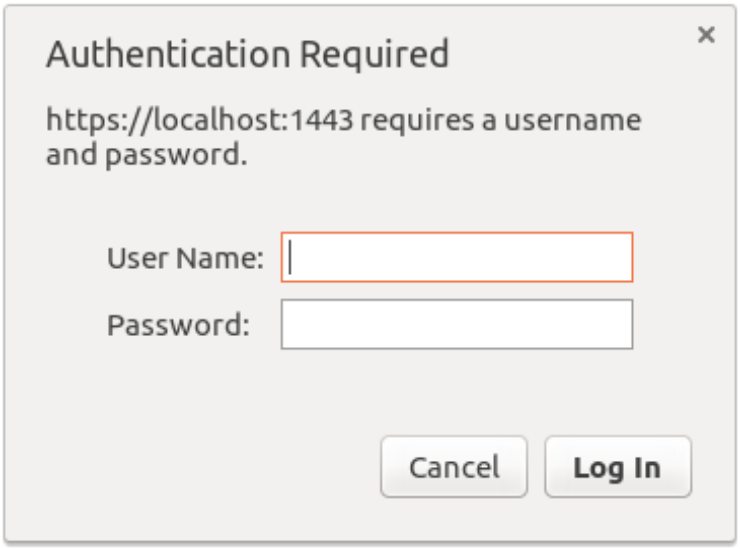

Figure 1 Basic access authentication in Chrome

After those fields are filled, the browser will send the request again with that data and the server can validate it. If data are correct, a response with Success code 200 is sent, otherwise Authorization Required code 401 is sent again. The browser will remember the name and password and keep them in a memory until window is closed. User doesn't need to put credentials with every request again

Even if the user fills the credentials only once, they are mandatory with every request, and the browser sends them every time. Users just does not have to think about it, however, developers have to. If the developer is creating a program, which will communicate with the API, he has to take care of sending appropriate authorization headers every time.

User's login and password are not encrypted, they are just encoded with Base64 algorithm. Encoding is needed for logins and passwords, which contains letters not allowed in HTTP headers. Even after encoding, for human it is not possible to read login and password, it is very easy to decode that data. If the developers need to secure user's credentials, they have to use safe transfer method as HTTPS with TLS.

Following examples show simple communication between the client and the server with simple authorization. The first request is from client to page /secure-area/. All headers are simplified, the actual looks is more complex. Only the parts related to authentication are shown here.

```
GET /secure-area/ HTTP/1.1
Host: spamler.rd.jst
```

The response from the server contains code 401 and WWW-Authenticate header, which informs the client that the server requires a name and password.

```
HTTP/1.1 401 Authorization Required
WWW-Authenticate: Basic realm="Spamler"
```

The second request is sent to the same URL as the previous one, but the Authorization header is added. After Basic word encoded name and password can be seen. The content before encoding by Base64 algorithm was Pavel:Password. The colon is a separator between name and password, which is also a limitation, because the name cannot contain a colon, otherwise it will be interpreted as delimiter.

```
GET /secure-area/ HTTP/1.1
Host: spamler.rd.jst
Authorization: Basic UGF2ZWw6UGFzc3dvcmQ=
```

3.4 REST

REST stands for Representational State Transfer and is one of the best practices in communication between an application and a server. It uses basic HTTP protocols and there are no special needs for supporting REST architectural style. Data are mostly transferred in JSON or XML format together with HTTP status codes, which defines success or errors of requests.

Individual resources are identified in requests using URIs as resource identifiers. The resources themselves are conceptually separate from the representations that are returned to the client. For example, the server does not send its database, but rather, some HTML, XML or JSON that represents some database records expressed, for instance, in Finnish and encoded in UTF-8, depending on the details of the request and the server implementation. (What is REST, 2016)

As REST is an acronym for REpresentational State Transfer, statelessness is key. Essentially, what this means is that the necessary state to handle the request is contained within the request itself, whether as part of the URI, query-string parameters, body, or headers. The URI uniquely identifies the resource and the body contains the state (or state change) of that resource. Then after the server does it's processing, the appropriate state, or the piece(s) of state that matter, are communicated back to the client via headers, status and response body. Most of us who have been in the industry for a while are accustomed to programming within a container which provides us with the concept of “session” which maintains state across multiple HTTP requests. In REST, the client must include all information for the server to fulfill the request, resending state as necessary if that state must span multiple requests. Statelessness enables greater scalability since the server does not have to maintain, update or communicate that session state. Additionally, load balancers don't have to worry about session affinity for stateless systems. (What is REST, 2016)

3.5 SMTP

Simple mail transfer protocol, or in short SMTP, is a standard for sending electronic mails. Mostly used ports are 25 for non-secure SMTP and 465 for secured SMTP also knows as SMTPS. As the protocol is text based, similar to HTTP, not ASCII characters are replaced with entities and attachments are encoded with Base64 algorithm.

Emails could be sent as HTML, plain text or both. Alongside with templates, emails can contain multipart content, which enables to the developer to send emails with HTML and also plain text message. This could be useful, if the recipient’s client does not support HTML messages, then they could see the message with HTML tags or unformatted, which could lead to ignoring or deleting the message.

3.6 HTML

HTML stands for HyperText Markup Language and is the most used language for the webpage development, based on Standard Generalized Markup Language, or shortly SGML. Even though the HTML was developed for webpages, nowadays developers could create applications with the HTML for Android, iOS or Microsoft Windows RT. Many new applications are developed with a web user interface because the development is faster and web browsers are running on every platform. The HTML itself is not powerful enough and is always used together with the CSS and JavaScript and even though, HTML5 is the newest standard of the language, the name mostly stands for a combination of those three technologies.

There are many frameworks for different programming languages, which are generating the HTML, however, the developer never works with the HTML directly. One of the examples is a JavaScript library React, which is described in the following chapter.

3.7 CSS, Sass and Compass

Cascading styles sheets or shortly a CSS is the language for describing a graphical appearance of the document written in markup languages like the HTML, however, it can be also used to style the XML document or the SVG document. The developer was able to style layouts, background colors borders and fonts with the older versions of CSS, however, the newest specification CSS3 enables to the developer use gradients, shadows and also animations. Even though all those improvements are a huge step ahead, front end developers are still missing some functionalities and they use preprocessors for the easier development like a Sass, Less or Stylus.

Sass

Stylesheets are getting larger, more complex, and harder to maintain. This is where a preprocessor can help. Sass lets you use features that don't exist in CSS yet like variables, nesting, mixins, inheritance and other nifty goodies that make writing CSS fun again. (Sass basics, 2016)

Sass provides many power tools to the developer, however, the output is still pure CSS because the web browsers does not understand the Sass or any other CSS pre-processor. In the first code snippet below is illustrated the Sass code, which uses variables, mixins and the nesting. The second code snippet shows the output from the Sass compiler, which transforms the Sass code into the pure CSS. The final code can be attached into the HTML document because the browser understands it.

```
$link-color: #8bb62b;
$link-hover-color: #96d30d;
$container-font-size: 16px;
@mixin clearfix(){
    &:after{
        content: '';
        clear: both;
        display: block;
    }
}
.container{
    @include clearfix();
    font-size: $container-font-size;
    a{
        color: $link-color;
        &:hover{
            color: $link-hover-color;
        }
    }
}
```

The output from the Sass compiler:

```
.container{ font-size: 16px }
.container:after{
    content: '';
    clear: both;
    display: block;
}
.container a { color: #8bb62b }
.container a:hover { color: #96d30d }
```

Compass

Even though the Sass brings many new features into the front end developing, there are many frameworks based on the Sass. One of the most known and also used for the Spamler development is the Compass, which brings more useful mixins and improvements to the Sass.

The Compass in the comparison to the Sass uses also a configuration file, where the developer can specify the setting of the compilation but also the folder structure. This is very useful for generating paths to the resources like images or fonts, because

they are mostly in the different directory than CSS files. If the paths to the assets are specified, more functions could be used. Compass except generating the relative or the absolute path also checks if the target file exists, otherwise the warning is printed out.

```
# Set this to the root of your project
css_dir = "www/css"
sass_dir = "www/css/src"
images_dir = "www/images"
fonts_dir = "www/fonts"
```

The part of the folder hierarchy of the Spamler application is described in the previous block as it is set in the configuration file of the Compass. The following code snippet explains the usage of two path related Compass functions, which are generating relative paths of the images and fonts. The pure CSS containing the relative paths is illustrated in the second code block.

```
@include font-face("OpenSans", font-files("opensans-regular-
webfont.woff2", "opensans-regular-webfont.woff", "opensans-regular-
webfont.ttf"));
h1.attachments {
    background: image-url('attachments-blue.png');
}
```

The output from the Compass compiler:

```
@font-face{
    font-family:"OpenSans";
    src: url('../fonts/opensans.woff2') format('woff2'),
         url('../fonts/opensans.woff') format('woff'),
         url('../fonts/opensans.ttf') format('truetype')
}
h1.attachments {
    background: url('../images/attachments-blue.png');
}
```

Another very useful function of the Compass is a sprite creator. The sprite is a big image, which consists of many smaller ones. If the sprite is used, the browser does not need to send a request for each image, however, only one request is sent. This technique decreases the needed time for loading all resources. Without the Compass, the developer has to join all images together manually, count coordinates of each subimage and write them into the CSS file for the big image positioning. If the Compass is used, the developer only has to specify which images he or she wants to include in the sprite and the Compass join them and counts coordinates automatically.

3.8 JavaScript

Neither the HTML nor the CSS are programming languages, the developer cannot write any logic in those languages and the JavaScript exists for the front end programming and scripting. Since more some applications in the web browser can supply desktop programs, JavaScript has become more important than ever. The great example could be Google Drive with Documents, Spreadsheets, Presentations and many more, which can replace desktop Word, Excel and PowerPoint from the Microsoft Office package.

When the importance of JavaScript is increasing, more frameworks and libraries are appearing. The client of the Spamler application uses more modern libraries together for the easier development and the possible future expansion.

React, React Router and JSX

The base of the application is written with the library React and the React Router plugin, developed by the Facebook Company. Comparing to the AngularJS, which is MVC framework, React is the library for building user interfaces and it contains only methods and functions related to the working with user components. The developer defines the components and their behavior and the React just takes care of their proper execution and rendering into the HTML output. The component could be nested into another component, which gives to the developer opportunity to separate the application into the smaller pieces, because they are easier to maintain.

The developer is able to specify different URL routes and attach components onto those routes with the React Router plugin, which takes care of rendering specified component in case of the URL is changed. With the router, the application acts like a regular web page with hyperlinks, however, the whole application is loaded and no redirections are performed. The code snippet below describe a simple React component, which counts how many times the button was pressed and prints it.

```
var ClickCounter = React.createClass({
    getInitialState: function() {
        return {clicked: 0};
    },
    click: function() {
        this.setState({clicked: this.state.clicked + 1});
    },
    render: function() {
        return (
            <button onClick={this.click}>
                Clicked: {this.state.clicked}
            </button>
        );
    }
});
ReactDOM.render(<ClickCounter />, document.getElementById("cnt"));
```

Please notice the HTML entry directly in the JavaScript. This is called a JSX format and was introduced by React developers for the easier HTML code writing with the React, however, this is not valid JavaScript and it must be converted into the valid code by the Babel library.

Flux, Event emitter and store

The Flux is also developed by the Facebook and is used as a dispatcher between the React components and the store. Even though React is not an MVC framework, the main idea of the MVC should be respected and the component, which represents the View, should not load any data or store them permanently and if the component needs a data, dispatcher should be used. The component should dispatch request with a payload specified the wanted data.

The store stores all data, loads them from the server and is registered to the dispatcher for a request receiving. If the component dispatches the request, the store receives it and performs the wanted action as loading data from the server or returns the data from its own storage. However, the dispatching implements only one way data flow and if store is ready to return data, it has to use the event emitter.

The event emitter is used for the opposite data flow than the dispatcher and also more components can register themselves for the events receiving and if store emits the event, all registered listeners receive the notification with the data passed to the event. An example of registering dispatcher and the request dispatching, registering events and the event emitting is illustrated in the following code snippet.

```
var DataStore = function() {
    this._data = null;
    Spamler.dispatcher.register(function(payload) {
        switch (payload.type) {
            case "loadAll":
                this._all();
        }
    }.bind(this));
    this._all = function () {
        // Load all into this._data
        emitter.emit("allDataReceived", this._data);
    }.bind(this);
}
var DataView = React.createClass({
    getInitialState: function() {
        return {data: null};
    },
    componentWillMount: function() {
        emitter.on("allDataReceived", this.dataReceived);
    },
    componentDidMount: function() {
        dispatcher.dispatch({ type: "loadAll" });
    },
    componentWillUnmount: function() {
        emitter.off("allDataReceived", this.dataReceived);
    },
    dataReceived: function(data){
        this.setProps({data: data});
    },
    render: function() {
        return <div>{this.state.data}</div>
    }
});
```

jQuery

jQuery is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers. With a combination of versatility and extensibility, jQuery has changed the way that millions of people write JavaScript. (What is jQuery, 2016)

Even though the jQuery also can manipulate with the DOM, in the combination with the React it is not easy to use, however, jQuery provides an easy way for executing AJAX request and deferred objects. Sometimes there are some inconsistences between browsers and the jQuery helps the developers to go through those obstacles.

```
var get1 = $.get(url1);
var get2 = $.get(url2);
$.when(getDetail, getExecutions).then(function(first, second) {
    // Success of get1 and get2
    // Do another request, based on the previous
    return $.get(first.LoadUrl);
}, function(error){
    // Error of get1 or get2
}).then(function (third) {
    // Success of 3rd reqest created in first success function
}, function (error) {
    // Error of 3rd request created in first success function
});
```

The deferred object is designed to avoid callback nesting, also called “callbacks hell”, by providing the possibility to chain multiple asynchronous executions as is illustrated in the code example above. All AJAX functions defined by the jQuery returns the Deferred object and the developer can use it directly.

3.9 Grunt

The Grunt is a JavaScript task runner written in the NodeJS language and the main purpose is to automate repetitive tasks which the developer has to perform. The Grunt itself cannot do anything, and the developer has to install plugins and configure them in the Grunt configuration file. With the Spamler the Grunt is used for running Compass and Babel compilation, however, more plugins are used.

Compass and PostCSS

For the CSS processing two packages are used, the preprocessor Compass and the postprocessor PostCSS. First all Sass files are compiled with the Sass compiler and saved as the CSS file, which is updated by the PostCSS task with the Autoprefixer plugin to include prefixed attributes for a better backward compatibility with older browsers.

Clean, Babel and Concat

For different JavaScript files slightly different tasks are run. If React JSX files are changed, the Clean task is run first to remove old compiled files, the Babel task is followed to compile JSX files into the JavaScript files. The Concat task is run to concatenate all JavaScript files into the single one and this task is run after all JSX files are processed, however, the Concat task is also run if only JavaScript files are changed.

Watch and BrowserSync

The developer can run specified tasks manually, however, with the Watch plugin, the higher level of automatization can be reached. The Watch plugin watches files and performs the specified actions if those files are changed. The developer can configure to run the Compass and the PostCSS tasks after any Sass file is changed and run the Clean, the Babel and the Concat tasks if any JSX files is changed. If only JavaScript file is changed, the task Concat is run. The BrowserSync task also watches files, however, if they are changed, the browser of the developer is refreshed and the developer does not have to wait until the compilation is done and do manual refresh. However, the Watch task watches source codes and performs compilation, the BrowserSync watches the output files.

4 Implementation

4.1 Three-tier architecture

The application is divided into three main parts known as three-tier architecture. The parts are Presentation tier, Application tier or Business logic tier and Data tier. Each part is created as separated Go package. Splitting an application into different packages is one of the best programming practices and brings many advantages for code maintenance or future expansion. Each tier communicates only with the tier underneath, which gives the opportunity to replace any layer without affecting others. The communication of layers is illustrated in Figure 2.

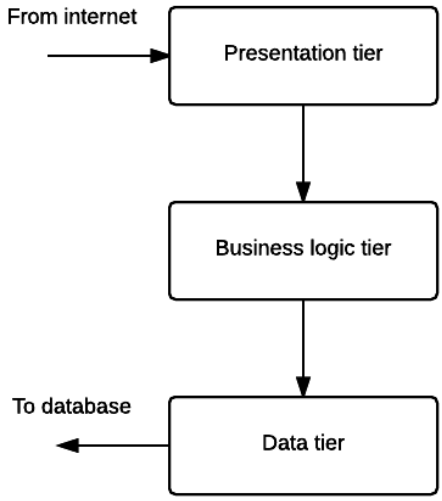

Figure 2 Three-tier architecture diagram

As seen on the diagram, Presentation layer can call functions of Business logic layer but not the other way round. Business logic never can call any methods of Presentation layer, the same is applied to Data layer and Business logic layer. If a program is scaled into layers, the developer has also the opportunity to split the program into different servers.

In the application the package of Presentation layer accepts and processes requests from the client. For requests Presentation layer parses JSON data into internal structures and for responses encode internal structures into JSON format. All functions of the layer are connected with routes described in a later chapter.

Business logic tier receives data from Presentation layer. Then it performs the requested actions, validates data and does all needed actions. Sending emails is also a part of this layer. The returned data after performing an action is encoded by Presentation tier again. The tier can call functions of Data tier if there are needs to load or save data to a persistent storage.

The lowest layer is Data tier which takes care of loading and saving internal structures persistently, in this case into SQLite database. In this package SQL queries and functions are defined for mapping data from internal structures into database or opposite way. If date and time are saved as integer in SQLite, this layer takes care of transforming between Go data types and SQLite data types.

If there are needs to change format from JSON to XML for example, only Presentation tier will be rewritten; the same applies to Data tier in case of changing from SQLite to another DBMS like MySQL or others, which is one of the major advantages of scaling an application into different layers.

4.2 Backend in Go language

The development started from the lowest level of the three-tier architecture, in a package called data, because there are no dependencies to the others packages, as is illustrated in Figure 2. In the data package are defined all SQL queries and the logic for communication with the database but also custom data types.

In the second phase, the middle layer of the three-tier architecture, the business logic tier, was developed; however, this takes place with minimum logic, just for passing the data between the data layer and the top layer, presentation tier. Additional logic and sending email were added to the business layer as the last step.

4.2.1 Configuration in TOML format

There are needs to configure the application and this data is mandatory during launching as URL for listening, the credentials of SMTP server or the location of assets. The format of the configuration file is TOML, which allows the administrator to include comments for a better orientation in the file or to explain the chosen settings.

Except the basic key-value pairs, TOML could contain nesting via tables, arrays or arrays of tables. However, the configuration file of the application does not use any nesting, only the top-level attributes. The following code snippet contains the part of the configuration file.

```
#### URL to listen ####
SecureURL    = "spamler.rd.jst:80"
NonSecureURL = "spamler.rd.jst:443"

#### Email sending ####
SMTP            = "mail.rd.jst"
Port            = 80
SkipInsecureTLS = true
```

As is visible, the value of an attribute could be string, number or Boolean. The datetime type is also supported, however, not used by the application. For parsing TOML files the applications uses the library made by BurntSushi, an example of parsing the previous snippet into the structure in Go language is illustrated in the following snippet. There must be a definition of the structure, which mirrors the schema of the file and the data are loaded by the library into this structure.

```
// Config object
type SpamlerConfig struct {
    SecureURL       string // SecureURL to listen
    NonSecureURL    string // Non secure URL for redirection
    SMTP            string // SMTP server
    Port            int    // Port of SMTP server
    SkipInsecureTLS bool   // Skip insecure SMTP server
}

// Load file
file, err := ioutil.ReadFile(filePath)
if err != nil {
    log.Fatal(err)
}
// Parse file
var configData SpamlerConfig
if _, err := toml.Decode(string(file), &configData); err != nil {
    log.Fatal(err)
}
```

4.2.2 Conditional build

Not all of attributes in the configuration file are mandatory, as the path to certificates for TLS or the assets folder, the path to the SQLite file or the initial SQL schema and many others, however, they are not specified explicitly, an implicit value must be assigned, as they cannot be empty and are mandatory for building and running. Unfortunately the initial values and the paths are different for debug builds and for release builds, however, Go language does not support conditional build like C or C++ using the macro system with a preprocessor.

Fortunately, the developer has opportunity to use a conditional building on the whole files using tags. They are implemented as comments and must be specified as the first line of the source code for the target file. During building, a few predefined tags exists for distinguish between the operating system and the architecture of the processor. More tags could be combined together by logical operators as and, or and negation. The developer can pass custom tags during the building by the command

line arguments as is shown in the following code snippet, when a custom tag debug-Build is passed.

```
go build --tags debugBuild
```

There is no possibility to set a value for the tag, its type is Boolean and only its presence can be tested. If the application is being built for debugging, the debugBuild tag should be specified, which will lead into the building with the file containing debug initial values for the configuration and the file with initial values for releasing is ignored. The first line of the next code snippet contains build tags for the debug build. Files containing this line are built, only if the debugBuild tag is specified during building as the command line argument. The second line is the opposite of the first one and is intended for the release build.

```
// +build debugBuild
// +build !debugBuild
```

If more build tags' lines are written together, then between those lines the logical operation AND is applied. The last build tags line must always be followed by an empty line, otherwise those lines are considered as a documentation comment for the following definition of package.

4.2.3 Gorilla and Mux router with Basic access authentication

Spamler is the application which is controlled via the REST interface and must work as the web service. Go language contains many packages for network I/O, however, some functions of those packages do not provide all needed functionalities, or it is hard to implement them.

The toolkit for Go language called Gorilla contains many useful packages, however, the application uses only a sub-package named Mux. The package implements a request router and dispatcher, and the main purpose of the router is to execute different functions for the given URL. Matching routes could be done as a simple text, a regular expression or a custom matching function. Another restriction is possible by HTTP methods, testing for present HTTP headers, a host and many others. If more

routes are prepended by the same part of the URL, the sub-routing could be used, which is also one of the best practices and is shown in the following code snippet.

Mux package is very powerful and also allows to implement Basic access authentication directly into the router, which is described in the code snippet below. All routes are defined on a sub-router returned by a matcher function containing the authentication logic. If the authentication fails, none of the routes defined on a secureRouter are executed and the execution ends in a NotFoundHandler, which sends HTTP Code 401 Unauthorized.

```
// All paths defined in secureRouter have to pass HTTP Basic auth
secureRouter := router.MatcherFunc(func(r *http.Request, rm
        *mux.RouteMatch) bool {
    name, pass, ok := r.BasicAuth()
    if ok && name == config.AuthName && pass == config.AuthPass {
        return true
    }
    return false
}).Subrouter()

// Campaigns routes
campaigns := secureRouter.PathPrefix("/campaigns").Subrouter()
campaigns.HandleFunc("/", ActiveList).Methods("GET")
campaigns.HandleFunc("/inactive/", InactiveList).Methods("GET")
campaigns.HandleFunc("/", UpdateOrCreate).Methods("POST")

// Will not pass HTTP Basic auth
router.NotFoundHandler = http.HandlerFunc(func(w
        http.ResponseWriter, req *http.Request) {
    w.Header().Set("WWW-Authenticate", `Basic realm="Spamler"`)
    w.WriteHeader(401)
    w.Write([]byte("401 Unauthorized\n"))
})
```

4.2.4 Server with and without TLS

One of the requirements was to secure the communication between the client and the server with TLS. The network package provides functions for a non-secure serving and also for a secure serving which needs a path to the private and public certificate. The non-secure serving is not important but it is better for the user experience. If the user forgets to use the https protocol and the server is not listening on the port 80, Connection refused error is shown to the user. However, if the server is also listening on port 80, it can redirect the user to the secure URL.

The code snippet illustrates the implementation from the application which supports listening also on the non-secure URL if the administrator specifies it in the configuration file. In case of accessing the non-secure URL, the user will be redirected automatically with HTTP Code 302 Found.

```
// Listen on HTTP and provide redirect (if non-secure URL is defined)
if len(config.NonSecureURL) > 0 {
    go func() {
        redirect := http.HandlerFunc(func(w
http.ResponseWriter, req *http.Request) {
            http.Redirect(w, req, "https://"+config.SecureURL+req.Req
uestURI, 302)
        })
        log.Error(http.ListenAndServe(config.NonSecureURL, redirect))
    }()
}
log.Fatal(http.ListenAndServeTLS(config.SecureURL, config.Certificate
, config.Key, router))
```

4.2.5 Templates rendering

Go language provides packages for parsing templates which are used for rendering the final content of the email with placeholders. As the email template could be in the plain text format or the HTML format, a different package must be used for parsing a different template type, the html/template package for HTML templates and the text/template package for plain text templates. Luckily both packages contain the same API and with a custom defined interface, the main logic could be implemented only once. All the template related entries are written into double curly brackets containing placeholders starting with a dot like .Recipient.Firstname, or a function call img as is illustrated in the code snippet below.

```
<p>Hello {{.Recipient.FirstName}} {{.Recipient.LastName}}</p>

How are you? Here is the picture of my cat <img src="{{ img
"cat.jpg"}}">

Regards, {{.Sender.FirstName}} {{.Sender.LastName}}
```

The content of templates is written by the user and saved in the database. Before each email sending, the template is parsed and placeholders are replaced by the given values as is described in the following code snippet. The first defined function is a custom templating function for replacing the img function call. To simplify the example code, only HTML parsing is shown.

```
var imgAttachments := regexp.MustCompile("\\{\\{([\\s-]*img )")

func replaceImgTagsInHTML(imgs []*data.Attachment, name string) (html
_template.URL, error) {
    for i := range imgs {
        if imgs[i].Name == name {
            return html_template.URL(imgs[i].Path), nil
        }
    }
    return "", fmt.Sprintf("Image attachment %v is not found", name)
}

func ParseTemplate(template string, attachments []*data.Attachment, r
ecipient *data.Recipient, sender *data.Sender) (string, error) {
    tplVars := map[string]interface{}{
        "Recipient":     recipient,
        "Sender":        sender,
        "ImgAttachment": attachments,
    }
    content := imgAttachments.ReplaceAllString(template, "{{$1.ImgAtt
achment ")
    parser, err := html_template.New("Email").Funcs(html_template.Fun
cMap{
        "img": replaceImgTagsInHTML,
    }).Parse(content)
    if err == nil {
        wr := bytes.NewBufferString("")
        err = parser.Execute(wr, tplVars)
    }
    if err != nil{
        return "", err
    }
    return wr.String(), nil
}
```

Please notice the string replacement in ParseTemplate function and a number of parameters for the replaceImgTagsInHTML function. In the first code example in the current chapter, only one argument is passed to img function call, the name of attachment. However the array of all attachments is needed to search for the given attachment by the name. To simplify the template writing for the user, automatic rewrite by the regular expression is made for the img function call. Even though the user does not need to specify the array name of attachments, before parsing the template, the name of the templates is inserted into the function call. The last code snippet shows the part of template before and after replacing.

```
<!-- Before replacing -->
<img src="{{ img "cat.jpg"}}">

<!-- After replacing -->
<img src="{{ img .ImgAttachment "cat.jpg"}}">
```

4.2.6 Shuffle slice

A random reorder a slice is needed for the list of recipients and the list of senders, because then, no pattern would be easily visible in the sent emails if they are not sent in the same order in each execution. Some programming languages contain a function for shuffling arrays like shuffle function in PHP or Python, however, many languages do not implement this function, and the developers has to write their own function.

The most known implementation is the Fisher-Yates shuffle, based on creating permutation of a finite set. Even though it is not hard to implement an own method for the random shuffle, it could be hard to do it properly. If a mistake occurs in the implementation, there is a possibility that some permutations are generated with a higher probability than the others, or in the worst case, some permutations are not generated at all. The Fisher-Yates shuffle produce an unbiased permutation, which means that each permutation is generated with the same probability.

However, the algorithm described below is a modern version of the Fisher-Yates shuffle algorithm designed for computer use by Richard Durstenfeld. The problem of the original algorithm is in its time complexity, which is $O(n^2)$, compared to $O(n)$ for the modern version. Two possible implementations of this algorithm exist, “in-place shuffle”, which does the shuffling in the original array and is described in the following code snippet, however, “inside-out shuffle” produces a new shuffled array and the original array stays unchanged.

```
func shuffleRecipients(shuffle []*data.Recipient) {
    rand.Seed(time.Now().UnixNano())
    for n := len(shuffle) - 1; n > 0; n-- {
        j := rand.Intn(n + 1)
        shuffle[n], shuffle[j] = shuffle[j], shuffle[n]
    }
}
```

4.2.7 Email sending with gomail

If the template is parsed successfully, the email itself could be sent. Go language contains the basic package for sending emails via the SMTP protocol, however, the developers have to build the whole message including headers according to the SMTP

specification by their own functions. This is not an easy task, especially if attachments are present in the message, which was one of the requirements for the application.

The decision to use an external library for building SMTP messages was made, and gomail package was selected, which wrap the existing SMTP package, however, it adds many useful functions for setting headers, recipients, sender but also attachments. The huge advantage of the package for the future expansion could be a possibility to write a custom sending function for a third side API and not using the SMTP protocol for sending emails.

In the following code snippet the whole message building and dialing SMTP server and sending email is desribed. The function uses ParseTemplate function, which was described in the previous sub-chapter, however, some inconsistence is visible according to simplifying those examples.

```
func sendmail(template *data.Template, attachments []*data.Attachment
, recipient *data.Recipient, sender *data.Sender) error {
    m := gomail.NewMessage()
    m.SetHeader("From", m.FormatAddress(sender.Email, sender.Name()))
    m.SetHeader("To", m.FormatAddress(recipient.Email, recipient.Name
()))
    m.SetHeader("Subject", template.Subject)
    for i := range attachments {
        m.Embed("/images/uploads/"+attachments[i].FileName,
           gomail.Rename(attachments[i].Name))
    }
    content, err := ParseTemplate(template, attachments,
        recipient, sender)
    if err != nil {
        return err
    }
    if template.Type == data.TYPE_HTML {
        m.SetBody("text/html", content)
    } else {
        m.SetBody("text/plain", content)
    }

    d := gomail.NewDialer(mailer.SMTP, mailer.Port,
        mailer.SMTPLogin, mailer.SMTPPassword)
    d.TLSConfig = &tls.Config{
        InsecureSkipVerify: mailer.SkipInsecureTLS
    }
    return d.DialAndSend(m)
}
```

4.2.8 Logging

Because all applications contain bugs, logging is a very important part of the program. It is very hard to fix some errors if the developer does not know, why and where they occurred. In those cases, logging errors help the developer more to understand, why the application did not execute correctly, or why it crashed. Logging into the syslog in JSON format was one of the requirements, and Logrus was found to be the most suitable package.

One of the advantages of Logrus is the possibility to specify more logging destinations, with different formatting for each of them. This is very useful during developing, while errors are sent in the JSON format into the syslog, however, errors are printed into the console as plain text, which is easier to read, and also some fields could be omitted.

4.3 Database

When the requirements had been finished, the database technology had not been decided yet, therefore, the first a logical model was made, as it is independent of a future database management system. The logical model contains general data types, as concrete DBMS nor their equivalent are known. Relations M:N between entities could be used as well, even the most of relations databases cannot create this relation directly. The logical model of the database is illustrated in Figure 3.

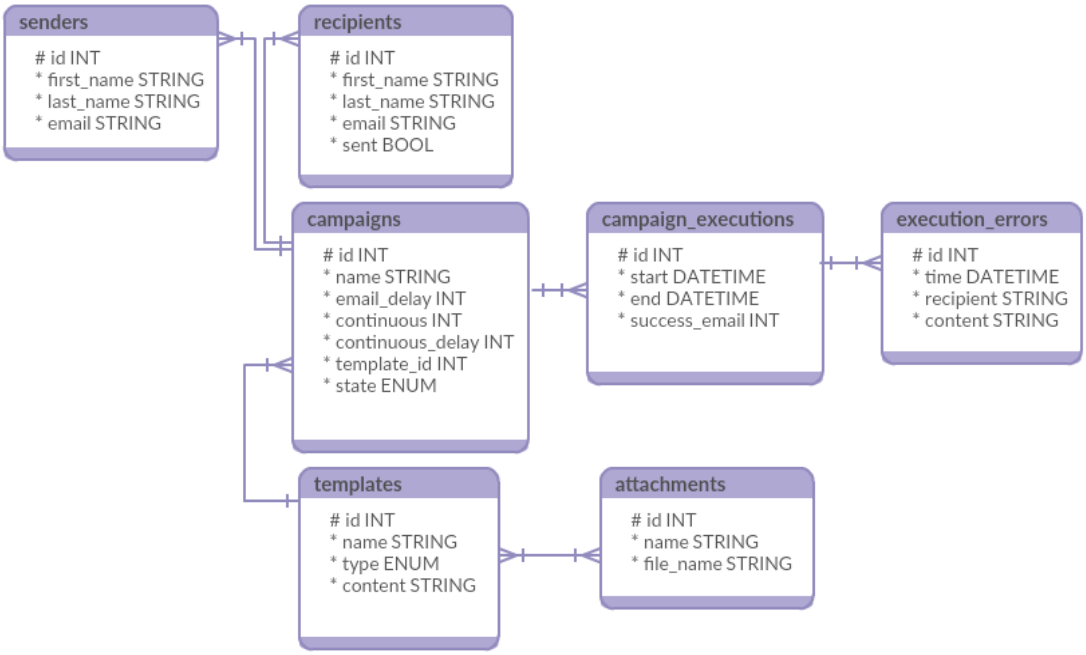

Figure 3 Logical diagram of database

Enum State in the entity campaigns could contain values Stopped, Paused and Running. Possible values in Enum type in the entity templates are HTML or Plain text. All relations must delete all descendants if a parent is deleted, except campaigns-templates relation. The template cannot be deleted at all if there is a relation with some campaign.

After the decision of using SQLite database was made, relation diagram could be created, based on the chosen DBMS. Limitations of SQLite forced to change some columns types to another types. For example, ENUM was replaced by INT with max length 2, even though two digits are not used in the current version of the program by campaign's state nor template's type. Boolean was also replaced by INT with max length 1, as only values 0 or 1 are saved. For datetime there were three options and UNIX Timestamp was selected, which is also saved in INT typed columns.

The final relation diagram is shown in Figure 4.

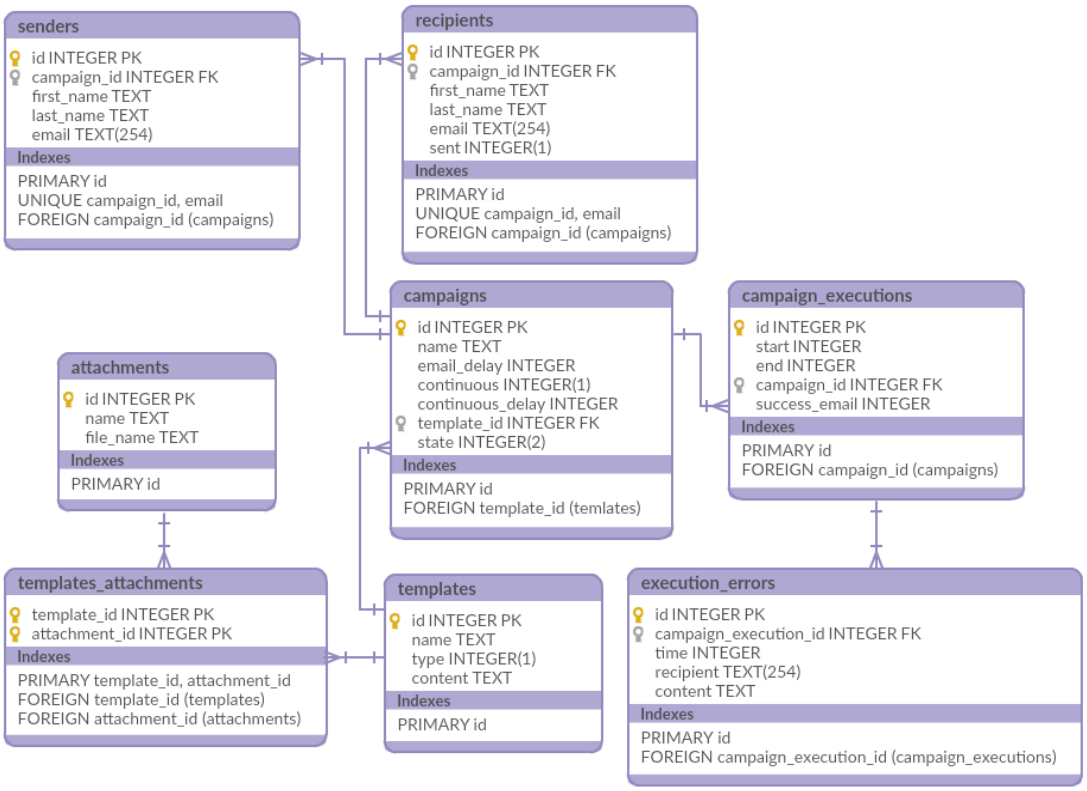

Figure 4 Relation diagram of database

4.3.1 Connection to SQLite in Go

Go provides database/sql interface for drivers, which provide unified communication with any database, even though some developers creating drivers are still not following this interface, which can lead into plenty of trouble in case of changing a database system in later times.

The only driver supporting the given interface found is go-sqlite3. In the following snippets an example of connecting to SQLite database is illustrated. The first PRAGMA query is to be noticed which turns on the foreign keys check. This is needed for backward compatibility with the older versions of SQLite and the developers have to specify if they want to use constraint checks.

```
import (
    "database/sql"
    _ "github.com/mattn/go-sqlite3"
)
func ConnectToDB(file string) error {
    db, err := sql.Open("sqlite3", file)
    if err != nil {
        return err
    }
    if _, err = db.Exec("PRAGMA foreign_keys = ON"); err != nil {
        return err
    }
    return nil
}
```

4.3.2 SQL injection and prepared statements

One of the most occurring SQL vulnerabilities of applications is a SQL injection, when the developer forgets secure users' inputs. If there are no checks of an incoming data from the users, a malformed code could be passed. In the following snippet an unsecure SQL query in Go programming language is shown, which allows the user to modify the query and execute own code.

```
login := "admin"
password := "' OR login = 'admin"

db.Exec("SELECT * FROM users WHERE login = '" + login + "' AND
password = '" + password + "'")
```

It is assumed that the developer does not escape inputs from the user and assigns values directly into the login and password variables. The final SQL query which is executed is shown in the next code snippet. The user does not know the password of an administrator, however, he or she can be logged with a full rights because of the SQL injection vulnerability.

```
SELECT * FROM users WHERE login = 'admin' AND password = '' OR login
= 'admin'
```

Two main possible techniques to prevent the SQL injection exist, an escaping and prepared statements. The escaping is more difficult for the maintenance as the developer has to take care of escaping each input separately.

Prepared statements were not mainly invented for the SQL injection preventing, the main goal of the prepared statements is increasing performance of the execution of the SQL queries. Each DBMS creates many execution plans and counts their I/O cost

and CPU cost before the execution of each SQL query, even those queries are the same. Analyzing and picking the cheapest plan took some time and for some queries this process could last longer than the execution itself.

If prepared statements are used, the DBMS builds, compiles and chooses an execution plan only once. The developer prepares statements for later use with placeholders instead of real values, so the query could be used repeatedly. When the execution is needed, the DBMS only loads the plan, replaces all placeholders by values and proceed the query. As the query is already built and compiled, the user cannot modify it by passing malformed code.

In the next code snippet the same scenario as in the previous one is shown, however, the prepared statement is compiled and executed separately. Even though the user enters the same data, now the SQL injection will not occur.

```
login := "admin"
password := "' OR login = 'admin"

stmt, err := db.Prepare("SELECT * FROM users WHERE login = ? AND
password = ?")
if err != nil {
    log.Fatal(err)
}
res, err := stmt.Exec(login, password)
```

4.3.3 No support for IN operator

Using of prepared statements is also one of the best practices, however, in SQLite driver for Go language support for more complex operators is missing. There were needs in the application to execute SQL queries with an IN operator, which accepts multiple values of the same type, however, those queries are not possible to prepare. Dynamic building of the query is required, which leads to the problem of SQL injection described in the previous chapter.

All the queries with the IN operator filter columns with the INTEGER type, which makes the task easier as there are no needs for escaping data. If the user sends other data than integers, the error of the mismatch types during decoding is returned and the query itself is not executed. In the following code snippet building and executing is illustrated.

```
func AddAttachments(templateId int, attachmentIds []int) (sql.Result,
 error) {
    glue := fmt.Sprintf("),(%d,", templateId)
    valuesSyntax := strings.Join(IntsToStrings(attachmentIds), glue)
    valuesSyntax = fmt.Sprintf("(%d,%s)", templateId, valuesSyntax)

    // Return the query in format ... VALUES (1,20),(1,21),...
    return db.Exec(fmt.Sprintf(`INSERT OR IGNORE INTO
        templates_attachments VALUES %v`, valuesSyntax))
}
```

4.3.4 Saving time

As described earlier, there is no DATETIME column type, and the decision was made to save the date and time as INTEGER in UNIX timestamp format. However, there is the difference between saving not set time in Go language and SQLite. Go language returns a negative number, however, SQLite returns 0.

If the date is not set, value 0 is saved into the database. The opposite procedure must be performed during loading the date from the database. In the following code snippets is the code for saving and loading the start date of campaign.

```
// Saving
var start int64 = 0
if !campaign.Start.IsZero() {
    start = campaign.Start.Unix()
}

// Loading
var start time.Time
if startFromDatabase > 0 {
    start = time.Unix(startFromDatabase, 0)
}
```

4.4 Frontend client

4.4.1 HTML and CSS

There is only one HTML file in the application because the whole frontend is written with the React JavaScript library. The only HTML present in the application is very short, and the whole HTML is described in the following code snippet.

```
<!DOCTYPE html>
<html>
    <head>
        <base href="/app/">
        <meta charset="UTF-8">
        <link rel="stylesheet" type="text/css" href="/styles.css">
        <script type="text/javascript" src="/scripts.js"></script>
    </head>
    <body>
        <div id="reactContainer">
        </div>
    </body>
</html>
```

The CSS code is divided into many files, however, all of them are defined as partial Sass files and they are included into a single file after the compilation. Because of this behavior, only one link tag is specified in the HTML and the file is downloaded once. It is better for the user experience because there are no delays caused by the loading of another file after visiting different parts of the client.

4.4.2 JavaScript

The whole logic, behaviors and the layouts of the client are defined by JavaScript with the libraries described in the chapter Technologies. The source code is divided into the many files for an easier maintenance, and the code block describes the folder structure. However, all JavaScript files are concatenated into a single one by the Concat task because the browser downloads one file faster than many of them with the same size.

```
js/
├─ jsx/
│    ├─ components/
│    └─ stores/
└─ src/
     ├─ components/
     ├─ plugins/
     └─ stores/
```

All scripts written in the JSX format are saved in the folder jsx and those files must be translated into the pure JavaScript files. Babel saves compiled files into the src folder and keeps the same folder hierarchy. The folder plugins contains source codes of all used libraries. The concatenate task joins only the files in src folder and saves the final code into the root folder and the browsers load only the final file.

4.4.3 Campaigns page

The campaigns page contains the list of active and the list of inactive campaigns illustrated in Figure 5. If the user accesses the application without any specific page, he or she is automatically redirected to the campaigns page. The user can start, stop, pause or delete the campaign in this page. There is a possibility to perform the actions to the single campaign or perform a bulk action to a group of selected campaigns. The name of the campaign works as a link to the list of all executions of the campaign.

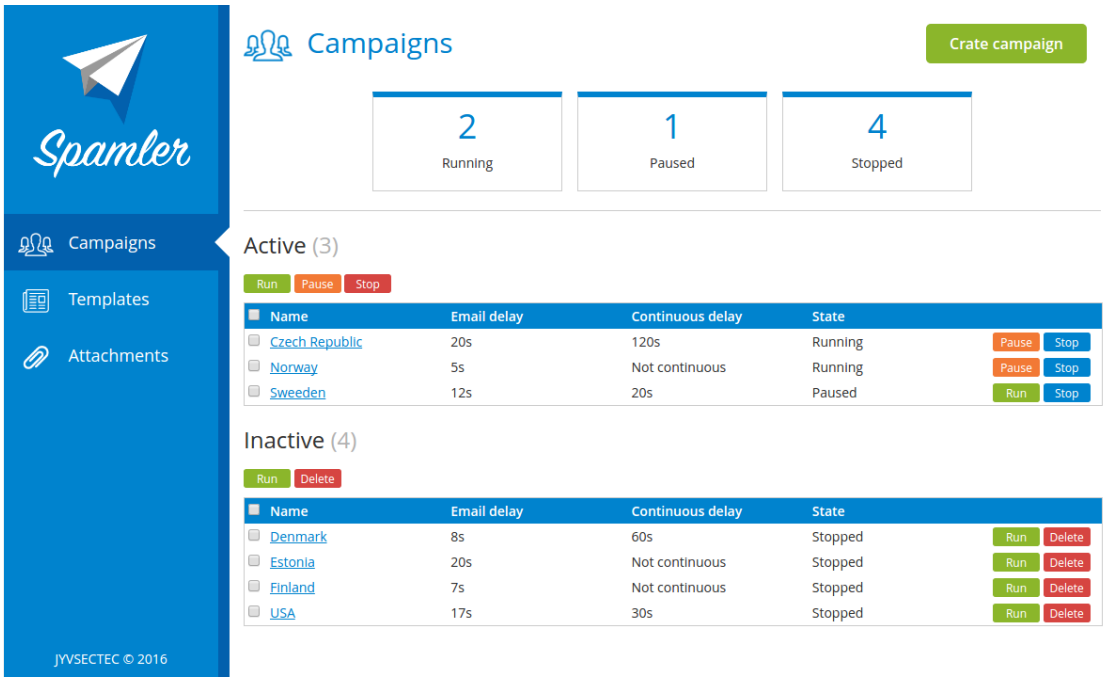

Figure 5 Campaigns page

Auto refresh

The whole page is automatically refreshed in the given interval, and if any changes occur in the data, they are visible to the user after the following refresh. This could be very useful if more users control the background service on separated computers. The automatic refresh notices the user about the changes without any actions. If the user performs any action and the list is refreshed, the auto-refresh timer is restarted to avoid often unnecessary requests to the server.

4.4.4 Campaign execution page

The campaign execution page lists all executions and errors of the campaign, and the user can perform any campaign action except the deleting on this page, which is illustrated in Figure 6. All executions ordered from the newest to the oldest are listed in the execution list and the selected execution is highlighted. The application prints all errors of the selected execution in the error list.

Figure 6 Campaign execution page

Auto refresh and selection of the newest execution

The auto refreshing functionality is also presents in the campaign execution page and it refreshes all the printed details in the given interval, except the errors of the older executions. Because there is no possibility how to update older executions or its errors, it would be wasting of time to refresh them. However, if the campaign is paused or stopped, the timer also stops its own execution. This could be a problem if another user starts the campaign, however, the application assumes that the user is not staying in the execution detail if the campaign is not running.

If the user starts the campaign from its detail, the application selects the newest execution automatically. Then after each refresh the user sees also all errors, even though he or she did not select the execution from the list.

4.4.5 Interactive response and notifications

If the server is busy, the time between the user performing the action and the response comeing could be long. If the time is really long the user can think that his or her action was not performed and he or she performs the same action again. To avoid this unwanted behavior, the application acts like the action is performed immediately. When the application receives the response, it shows the notifications illustrated in Figure 7. After all responses are received, the application refreshes the view to reflect the current state of the application.

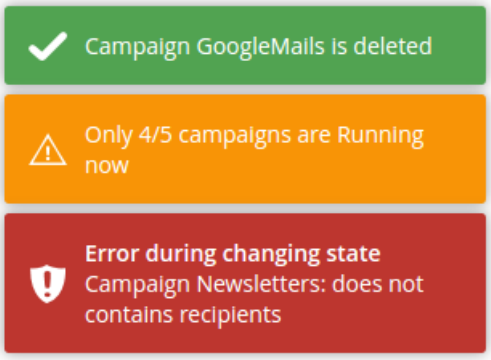

Figure 7 Notifications

4.4.6 Errors from server

If the server returns an error, there are many possible ways how the client application can react. If the error interrupts the performing of the campaign action, the application only shows notification as described in the previous chapter. However, if the application receives an error instead of the data, it prints the message to the user as is illustrated in Figure 8.

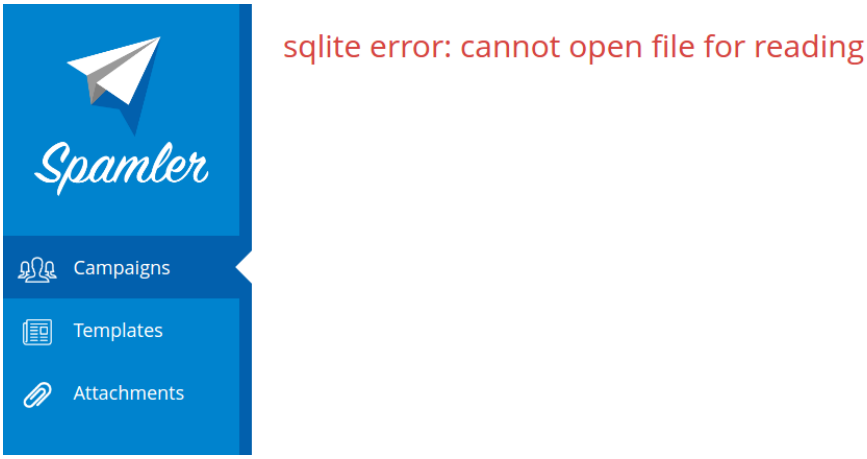

Figure 8 Error from server

Partial error from the server

Sometimes the data for the view are loaded by many requests and only some of them could fail. In that case, the application can render the view and prints the error as a part of the view. See Figure 9, when only the request loading the execution errors failed.

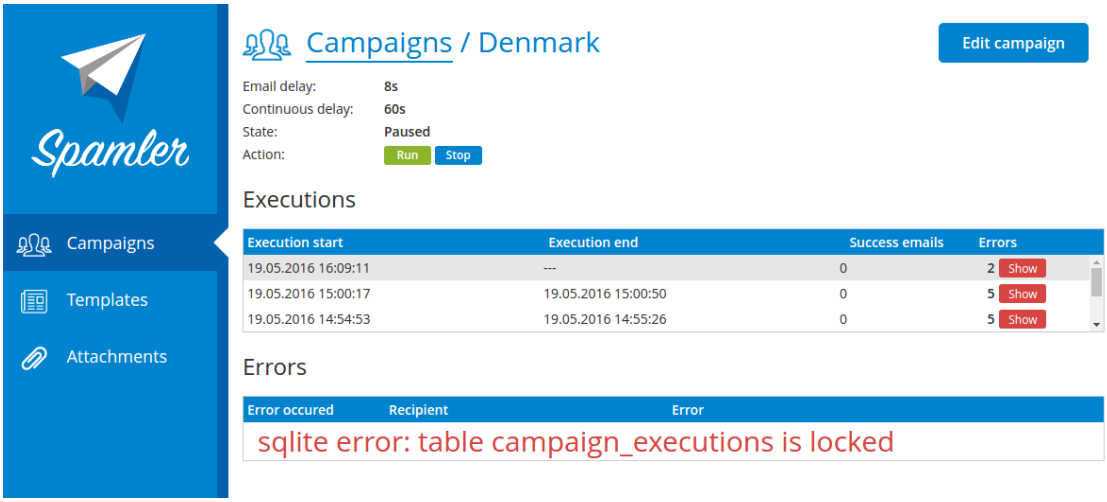

Figure 9 Partial error from server

4.4.7 Server unreachable page

The errors described in the previous chapters are returned by the server, however, the request could fail due to the refused connection when the server is not working or the internet connection between the client and the server is not available. If the server is unreachable, the application replaces the view by the error message and the

countdown shows the remaining time until the next attempt to reach the server as is illustrated in Figure 10.

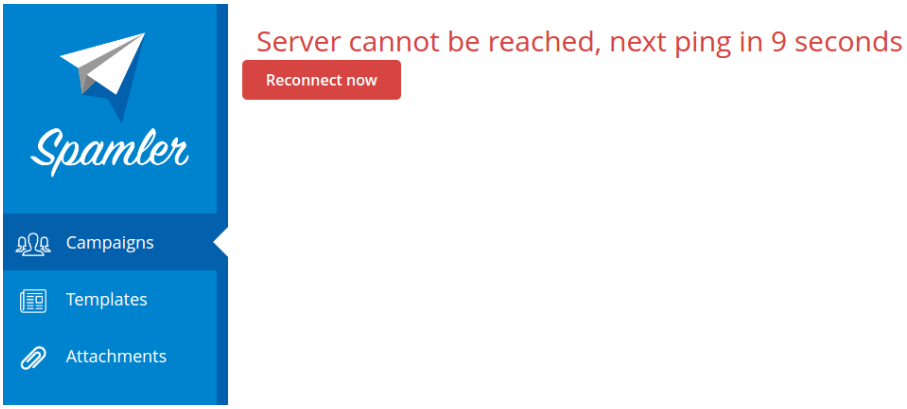

Figure 10 Unreachable server

The application tries to reach the server again after the given delay and if the server is still not available, the countdown starts again from the beginning. This scenario is repeated until the connection is established. However, if the user thinks the server is reachable and he or she does not want to wait for the next attempt, he or she can use the Reconnect now button to check the server availability immediately. If the response is successful, the applications renders the previous view again and the user can continue on the same place where the interruption was.

5 Results

The aim of the thesis was to develop the application to generate email traffic. The thesis describes all technologies and techniques, which were used in the development of the server part and the client part. Even though the server part and the API are fully functional, the client is still in the development. The graphical user interface intended for mange templates and attachments is not available at all. However, the administrator is able to use the API to manage templates and attachments and controls campaigns in the client.

The first part of the thesis describes the requirements set by the assigner, followed by used technologies during the development and implementation of the application. The Go language is used as programming language of the server part together with SQLite as a persistent storage and the SMPT protocol to sending emails. The application is configured in the configuration file in the TOML format. Users and clients can communicate with the server via the REST API in a JSON format secured by the TLS and the HTTPS protocol. The access to the server is restricted with the HTTP Basic authentication, and the user has to know the name and the password if he or she wants to control the server.

The client is styled with CSS and is developed in JavaScript with the React library for rendering the views, Flux dispatcher and Event emitter are used for communication within the components and the data stores. The thesis also describes the usage of the Grunt, Compass, Autoprefixer and Babel and their advantages for the front end development.

The application is not finished now and the template part and attachment part must be resolved, nevertheless, there are is also room for improvement. For example, it would be useful for the user to see in the campaign list, how many successful emails were sent and how many errors occurred during the last execution of each campaign. However, the priority was to create a functional application.

6 Conclusion

I learned about many new technologies and techniques, which I did not know before. Even though I have done the web page development before, I never used the Go programming language nor the JavaScript library React. I also tried to use popular development methods and improve my skills with techniques as the Compass, Babel and Grunt.

Alongside what I learned in the web development field, I also got new experiences with the Linux operation system. Because all servers of the JYVSECTEC Company are based on Linux, it was easier to use Ubuntu distribution on my computer.

All the programming and scripting languages I used are very modern and I think the knowledge of them could help me in my future career. My supervisor also required from me to write a clean and commented source code and I got great programming skills.

References

About JYVSECTEC. Page on JYVSECTEC webpage presentation. Accessed on 25.04.2016 Retrieved from http://jyvsectec.fi/en/about-us/

About SQLite. Page on Official SQLite website. Accessed on 28.4.2016 Retrieved from https://www.sqlite.org/about.html

Cyber environment. Page on JYVSECTEC webpage presentation. Accessed on 25.04.2016 Retrieved from http://jyvsectec.fi/en/cyber-environment/

Datatypes In SQLite Version 3. Page on Official SQLite website. Accessed on 28.4.2016 Retrieved from https://www.sqlite.org/datatype3.html

Go programming language. Page on official Go documentation website. Accessed on 28.4.2016 Retrieved from https://golang.org/doc/

Sass basics. Guide of Sass preprocessor. Acessed on 15.5.2016 Retrieved from http://sass-lang.com/guide

What is jQuery. Official page of jQuery library. Acessed on 20.5.2016 Retrieved from http://jquery.com/

What is REST. Lessons of REST api tutorial Accessed on 30.4.2016 Retrieved from http://www.restapitutorial.com/lessons/whatisrest.html