VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Aleksi Gold

INTEGRATING A NAMESERVER INTO

A

MODULAR SERVICE PLATFORM

Information Technology
2018

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

TIIVISTELMÄ

Tekijä:	Aleksi Gold
Opinnäytetyön nimi:	Nimipalvelimen integrointi modulaariseen palvelualustaan
Vuosi	2018
Kieli:	englanti
Sivumäärä:	85
Ohjaaja:	Ghodrat Moghadampour

Opinnäytetyön tavoitteina oli valita ilmainen ja avoimella lähdekoodilla oleva nimipalvelin, sen konfigurointi kehitys- ja Openshift ympäristöön sekä nimipalvelin integrointi jatkokehityksenä modulaariseen palvelualustaan. Integrointi koostui asiakasohjelmasta, joka mahdollisti kommunikaation nimipalvelimen kanssa, sekä käsittelijästä, joka tuki nimipalvelimen tietueiden listausta, luomista, poistoa ja muokkaamista. Käsittelijän tukemat toiminnot piti esittää käyttäjälle hyvällä käyttökokemuksella sekä modernilla käyttöliittymällä.

Yllämainittujen tavoitteiden lisäksi opinnäytetyöhön kuului uuden palvelun luonti olemassaolevasta palveluresurssista, sekä yksittäisten palveluresurssien hallinnointi käyttäjän puolesta. Käyttäjien toiminnan lokitus toteutettiin suunnittelemalla uusi taulu tietokantaan, sekä lisäämällä käsittelijöihin tarvittava toiminnallisuus.

Nimipalvelimeksi valikoitui PowerDNS, jossa oli valmiina tarvittavat rajapinnat. Integrointi toteutettiin käyttämällä Java 8:aa ja käyttöliittymässä käytettiin JavaScriptiä ja siihen tarkoitettua React-kirjastoa. Toteutus tehtiin käyttäen nelivaiheista prosessia: (1) tavoitteiden määrittely, (2) toteutuksen suunnittelu, (3) toteutus ja (4) testaus.

Testausta suoritettiin koko projektin ajan käyttämällä yksikkö-, “päästä päähän”- ja käyttäjätestausta. Näin ollen projektin onnistumista voitiin mitata ja kaikki tavoitteet saavutettiin.

Avainsanat DNS, Java, JavaScript, REST

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author:	Aleksi Gold
Title:	Integrating a Nameserver into a Modular Service Platform
Year	2018
Language:	English
Pages:	85
Name of Supervisor:	Ghodrat Moghadampour

The objectives of this thesis were to choose a free and opensource Domain Name Service (DNS) server, configure the server for local development and it had to be able to run in OpenShift. A client for communicating with the DNS server had to be implemented together with a handler for handling the listing, creation, deletion and modification of DNS zones. The actions supported by the handler had to be presented to the user with a good user experience as well as a modern user interface.

In addition to the objectives above, other objectives included in the project were the ability to create new services from existing resources as well as manage resource on the behalf of customers if need be. A way of logging actions also had to be implemented as well as presenting individual resources with management capabilities for the user. All the mentioned objectives were achieved during the project.

The DNS server selected was PowerDNS, which provided a REST API for interfacing. The integration was developed using Java 8 and the frontend was developed using JavaScript and React.

Development was carried out using a 4-step process: (1) defining requirements, (2) designing solution, (3) implementation and (4) testing, which was performed throughout the project in the form of unit, end-to-end and user testing as well as at the end of the project with all project requirements completed.

Keywords DNS, Java, JavaScript, REST

CONTENTS

LIST OF FIGURES AND TABLES

LIST OF ABBREVIATIONS

MSP	Modular Service Platform
DNS	Domain Name System
ISP	Internet Service Provider
TTL	Time to Live
REST	Representational State Transfer
LDAP	Lightweight Directory Access Protocol
SPA	Single Page Application
E2E	End-to-end
BIND	Berkeley Internet Name Domain
ISC	Internet Systems Consortium
VCS	Version Control System
GNU	GNU is not Unix
GPL	GNU General Public License
HTTP	Hypertext Transfer Protocol
POJO	Plain Old Java Object
URL	Uniform Resource Locator
API	Application Programming Interface
SQL	Structured Query Language

1 INTRODUCTION

Currently there is an abundance of companies that offer multiple internet related services that can be administered from the comfort of our own offices or even sofas. These services usually have a dashboard that may be accessed from a web browser to manage the offered products.

The objective of this project is to integrate a Domain Name Service (DNS) server and other improvements for a Modular Service Platform (MSP) as a continuation of an earlier thesis project by Niko Lehto at Jubic Oy. MSP is a hosting platform that can be integrated into any external system. Existing integrations include an issue tracker, Redmine, and OpenShift, a cluster for hosting Docker applications. /1/

Jubic Oy is a systems and software development as well as a hosting company located in Vaasa, Finland. It is a moderately new company and currently employs four full-time and four part-time employees. The company was founded in 2014 by two students while studying Information Technology at Vaasa University of Applied Sciences. /2/

This document provides an overview of the development of the integration between two already implemented systems, including how unit and end-to-end testing are implemented to ensure that the integration continues to work in the future.

The project was developed using Java 8 for the backend and JavaScript, HTML and CSS for the frontend. The two components communicate with each other over a RESTful API.

2 RELEVANT TECHNOLOGIES

2.1 Domain Name System (DNS)

The Domain Name System (DNS) is a decentralized system which could be compared to a phone book. Instead of names of people being mapped to phone numbers, domain names, for example “localhost”, are mapped to IP addresses, for example “127.0.0.1”. Instead of having to remember IP addresses, a user only need to know the easier to remember domain name.

DNS is decentralized as a user may host the DNS records of their domain, for example “example.com” themselves if they choose to. This kind of DNS server is called an authoritative nameserver. The address of these servers must be set for the top-level domain (TLD) nameserver. TLD nameservers are represented by the leftmost part of a domain, for example “.com”. Furthermore, the TLD nameserver must be set at root nameservers. These are the topmost nameserver in the hierarchy.

There is also a fourth kind of DNS server called a DNS recursor. The task of this kind of server is to cache and serve DNS requests to most clients that access DNS records. They help offload work from the various DNS servers and may speed up DNS queries if somebody else has recently accessed the same records that a client requests. Most ISP’s offer a DNS recursor for their customers. /3/

For example, if one were to type www.kulta.us in a browser on a PC with no cached DNS information, a request would be sent to the DNS recursor configured on the PC. In the worst-case scenario, the recursor does not have any cached records related to any part of this address. This means that the recursor would query the root nameservers for information about where the TLD nameservers for the '.us' domain may be found. After a response is received, the recursor would query the TLD nameserver for information about the nameservers of 'kulta.us'. After this, the recursor can finally query the nameserver which holds information for the address www.kulta.us and an IP address would be returned to our PC.

DNS is a very important part of the internet ecosystem and makes accessing websites and servers easier by working in the background. Most internet-connected devices use DNS to contact other devices over the internet.

2.2 Java

Java is a programming language and ecosystem originally developed by Sun Microsystems in 1995, and later acquired by Oracle. The Java ecosystem consists of the Java Virtual Machine (JVM), core classes and the compiler. The language is a strongly-typed object-oriented language that is compiled to bytecode, which is interpreted by the JVM. The JVM also abstracts away memory management from the developer unlike languages, such as C or C++. A major benefit of using Java for developing applications is that the compiled bytecode is not platform dependent. Rather, it supports any platform that has a JVM implementation available. The developer may write code once and be able to run the program on most devices without having to worry about which operating system the user has. /4/

2.3 Docker

Docker is an open source platform for building and deploying applications that have been packaged together with all needed dependencies in a reproducible fashion. A Docker image is created by writing a “Dockerfile”, containing every step executed for the final image. The file starts by inheriting a base image, which most likely is a “Dockerized” operating system. The steps are cached so that they do not need to be executed every time something changes in the file. The final image may then be distributed in to any environment and should work similarly to a local development environment as the image is the same. Docker images run in containers which are isolated processes which is another benefit of using Docker, due to separation of concerns and added security. /5/

2.4 Hypertext Transfer Protocol (HTTP)

Hypertext Transfer Protocol (HTTP) is a stateless protocol used since 1990. The most resent version of HTTP is HTTP/2. However, the most used version is HTTP/1.1. HTTP works over TCP connections via requests and responses. Web browsers and many other applications use HTTP to communicate with other internet-connected devices. /6/

HTTP works via a client requesting a resource, such as a HTML file, from a server, which may then respond with the contents of the file. For example, if you access a website from a web browser, the browser will initially send one request for the HTML on the server. The HTML may contain links to JavaScript and CSS files which are then each fetched with a request per file.

2.5 Representational State Transfer (REST)

Representational State Transfer (REST) is an API architectural style for communication between different programs utilizing HTTP for transport. REST APIs commonly support 4 kinds of HTTP methods; GET, POST, PUT and DELETE and data is usually serialized as XML or JSON. /7/

RESTful are accessed with an URL together with a HTTP method and optional content in the body of the HTTP request. For example, the GET method to the address “/api/people”, could return a serialized list of people that the program knows about. Furthermore, a GET request to “/api/people/1”, could return a single person that has an id of 1. A POST request usually creates a new entity, while PUT updates an existing entity and DELETE deletes an entity.

As may be observed, RESTful web services are quite simple to grasp and debugged by developers, while other solutions such as SOAP or XML-RPC are more complex, however a downside to REST is that there is no well-defined standard and implementations most likely differ for each application.

2.6 JavaScript Object Notation (JSON)

JavaScript Object Notation (JSON) is a universal data serialization format based on the JavaScript object format. A major strength of JSON is that it is easy for both developers and machines to parse and utilize and most languages provide a library for serializing and deserializing JSON. /8/

The idea of JSON is that it is an actual subset of JavaScript. Valid JSON is also a valid JavaScript object. A simple example of JSON may be observed in Code Snippet 1 below.

```
{
  "id": 1,
  "name": "Aleksi Gold"
}
```

Code Snippet 1. Example of JSON

As seen in Code Snippet 1, JSON is very lightweight. The text represents an entity with an id attribute of 1 and a name attribute of “Aleksi Gold”. This example could be sent from a backend application programmed in Java and received by a frontend application programmed in JavaScript and both languages are able to understand and deserialize it into a native object. JSON can also be stored in databases, such as MariaDB.

2.7 Dropwizard

Dropwizard is a collection of Java libraries that together provide a way of creating high performance web applications. The libraries powering Dropwizard include: Jetty, Jersey, Jackson, Logback, Hibernate Validator, Metrics, JDBI, Hibernate and Liquibase. Dropwizard is released under the open source Apache 2.0 license. /9/

Dropwizard is used for creating RESTful web services that communicate with a database, such as MySQL. While Dropwizard uses good and known libraries that could be used separately, the main benefit of Dropwizard is that the libraries have been configured and tested to work seamlessly together to help focus on developing high performance web services.

2.8 MariaDB

MariaDB is a popular open source relational database which was forked from MySQL by the original developers in 2009. It consists of the server itself as well as binaries used to query and administer the server. Data is stored in many different formats, including: text, bytes or integers which can be queried using a dialect of SQL. /10/

Most dynamic websites, as well as other applications, use a database like MariaDB or MySQL to store user data. This data is then queried using SQL to find the needed information and may then be presented to the user. Data is stored in tables as individual rows. During querying, data between tables may be joined to form a new table of rows with the wanted data. An example of a SQL query is presented in Code Snippet 2.

```
SELECT * FROM user WHERE id = 1 LIMIT 1;
```

Code Snippet 2. Example of a SQL query

In Code Snippet 2, the “users” table is queried for a single user with the id of 1. The query can be broken in to five logical parts. Firstly, the “SELECT” keyword is used for querying results, while “INSERT”, “UPDATE” and “DELETE” are used to create, modify and delete data. Secondly, the asterisk in the example query is used to define that all columns are requested. Thirdly, the “FROM” keyword together with a table name, in this case “user” is used to define which table to query. Fourthly, the “WHERE” together with a condition is used to filter the rows for only the results wanted. Finally, the “LIMIT” together with an integer is used to specify that only the first “x” results are wanted.

2.9 jOOQ

jOOQ is a Java library intended for generating database handler code against existing schema in a database. The generated code is reflected against the types in the database and dialect specific SQL is abstracted away. The methods for building queries resemble SQL but are chainable Java methods. /11/

The generated POJOs and handlers are defined in the configuration of the Java project together with the Java package that the code will be placed in. Using the generated jOOQ classes and methods, data in the database can be queried in to POJOs which can then be further processed. Using jOOQ keeps our database design and access simple and maintainable. The corresponding jOOQ query to the SQL in Code Snippet 3 is presented below.

```
DSL.using(configuration)
        .select()
        .from(USER)
        .where(USER.ID.eq(1))
        .fetchOne();
```

Code Snippet 3. Example of a jOOQ query

The resulting record from the query above would result in the same data as the earlier SQL example, except that the data is accessible from a Java object. Using jOOQ makes storing and querying a database from Java easier and safer that manually constructing SQL queries and using the built-in database libraries.

2.10 Liquibase

Liquibase is a Java library intended for define database schema using XML, YAML, JSON and SQL. The library eases multiple developers to work on database schema using branching and merging and may be executed at startup or during the build to ensure that the application is running against the latest database schema. /12/

Liquibase defines schema changes in "changesets" which are executed in order resulting in a reproducible database schema. A simple changeset defined in JSON is presented in Code Snippet 4 below.

```
{
  "changeSet": {
    "id": "1",
      "author": "Aleksi Gold",
      "changes": [{
        "createTable": {
          "tableName": "user",
          "columns": [{
            "column": {
              "name": "id",
              "type": "BIGINT",
              "autoIncrement": true,
              "constraints": {
                "primaryKey": true,
                "nullable": false
              }
            }
          }, {
            "column": {
              "name": "name",
              "type": "VARCHAR(100)"
            }
          }]
        }
      }]
    }
  }
```

Code Snippet 4. Example of a Liquibase changeset

In Code Snippet 4, a table called “user” is created in the database with two columns. The first column is called “id” and is of the type “BIGINT”, a 64-bit integer. This column is used as the primary key of the table; cannot be “null”, which means that it cannot have an empty value; and will be incremented for each new row. The second column is called “name” and is of the type “VARCHAR(100)”, which means that it can store a variable length piece of text with a maximum length of 100.

2.11 OkHttp

OkHttp is HTTP client built by Square for Android and Java programs. It supports modern HTTP features, such as HTTP/2, GZIP and modern TLS. The library is licensed under the open source Apache 2.0 license. /13/

While Java does provide a HTTP client, it is quite old and does not support many of the features that OkHttp does. OkHttp makes communicating via HTTP easier and provides builders for configuring the client, in a clean and readable way, to perform as expected.

2.12 Jackson

Jackson is Java library which provides the ability to serialize and deserialize between Java objects and JSON using the “readValue” and “writeValue” methods found in the ObjectMapper class. /14/

Jackson provides Java annotations for specifying the mapping between a Java class’ property and a JSON property. A Java object is serialized by providing it as a parameter to Jackson’s “writeValue” method, which returns a string containing the corresponding JSON. Deserializing JSON works by providing the “readValue” method the JSON as a string accompanied by the Java class which it should be deserialized to. An example of a Jackson annotated POJO is presented in Code Snippet 5.

```
class User {
   @JsonProperty
   private Long id;
  @JsonProperty
   private String name;

   public Long getId() {
      return id;
   }

   public String getName() {
      return name;
   }

 public void setId(Long id) {
      this.id = id;
   }
```

```
        public void setName(String name) {
          this.name = name;
        }
      }
```

Code Snippet 5. Example of a Jackson annotated POJO

The class in Code Snippet 5 may be serialized in to the JSON example previously presented. Properties with the “JsonProperty” annotation are serialized, and deserialized using the getter and setter methods. Jackson makes it straightforward to convert data between objects and JSON.

2.13 Cascading Style Sheet (CSS)

Cascading Style Sheet (CSS) is a language used to define the look of HTML and XML documents. While CSS does not affect the content of documents, it describes the way the document should be presented to the user. CSS is an important language in the web and has been standardized for all browsers. /15/

While CSS is not essential for delivering information to users, it can make the information more readable and therefore is of great significance in the modern web. CSS is used to personalize the look of a website and without it most website would look very similar.

2.14 JavaScript

JavaScript is a programming language mainly used in web browsers. The most notable non-browser use of JavaScript is NodeJS. JavaScript is classed as a dynamic scripting language as the source code is interpreted at runtime instead of compiling ahead of time, like other languages, such as Java. /16/

While browsers use HTML for content and CSS for presentation, JavaScript is the third building block of websites and is used to provide functionality to otherwise static websites. JavaScript is also an important part of the modern web and some websites do not function properly if JavaScript is turned off in the browser. JavaScript can, for example, be used to fetch content from a server in the background and dynamically update the content of a website without the user having to do anything.

2.15 Single Page Application

Single Page Applications are web applications which initially load a single HTML file and dynamically change content based on user interactions. SPAs load content in the background and manipulate the document object model. A SPA can look and act like a normal web application, with the exception that the content is only update partially. /17/

SPAs are a modern way to create web applications. While the initial page load must download a large amount of JavaScript, the following requests will be smaller and faster as they will most likely only be JSON objects with new data. As only new data is loaded in the background the user interface will act much faster with mobile and high latency connections.

2.16 Bootstrap

Bootstrap is a CSS ja JavaScript framework developed by Twitter, which provides readymade stylesheets and components for developing a modern and responsive web user interface. Bootstrap makes it easy for developers to create websites that work on various screen sizes. /18/

With the web being accessed by competing browsers, which have their own implementations of JavaScript and CSS, it has become hard to write code that looks and works the same across all browsers. By using Bootstrap, the workload of a developer creating a modern and responsive website is reduced significantly. The developer can customize the look and feel of the website while still relying on the tested cross-browser supported framework.

2.17 React

React is a popular JavaScript framework developed by Facebook and was initially released in 2013. React is used to create reusable components for single page applications. The library provides an interface for binding data to HTML and when data changes, only the part of the user interface, which changes is repainted. This results in performance gains versus most implementations of dynamic content changes. /19/

React allows developers to create components and use them across a web application when needed. A single JavaScript file can contain the CSS, HTML and JavaScript parts of a component with all of it written in JavaScript. Each component can have its own state which can affect the presentation or functionality of the component.

3 APPLICATION DESCRIPTION

MSP consists of a server application, written in Java, with the key libraries being Dropwizard and jOOQ. The server also serves the web client that utilizes React for a "Single Page Application" experience. These two components communicate with each other using RESTful web services. The third key component of the application is the database, which is MariaDB, a fork of MySQL.

The architecture of the application is designed in a way that any external systems may be integrated with it via implementing the handler interface for each system, which defines custom actions that it supports. These actions are sent to a single endpoint with the action type, provider, target id and user data. Each handler is then looped through and the handler makes a decision to support the requested action or not. Another task of the handler is to provide data on the supported component, which can then be shown in the web client.

In this project's case, MSP is being used as a hosting platform and it already has two handlers implemented, OpenShift and Redmine. One of the objectives of this project was to implement a new kind of service resource for DNS via another handler. This consists of choosing a DNS server, creating a client and handler, defining actions and creating a frontend component for the DNS service resource. The functionalities of this handler must be tested thoroughly using automated testing techniques, such as E2E, integration and unit testing methods.

Another important objective is that administrators should be able to create services and manage existing resources of all customers, utilizing the administrator view and their own LDAP credentials. This functionality can also be reused for allowing customers to view their individual resources, allowing single DNS resources without other resources to be shown and managed without the need to show a service for it.

Lastly, one of the objectives is to design a way of monitoring actions on services and their resources for keeping track and automating the billing of customers in the future.

3.1 Requirements

The requirements of this project may be categorized, according to priority, into three different categories. These categories are “Normal”, “Expected” and “Extra” requirements, with the first category being of most importance and the last of least importance. The categories and their requirements are presented below.

3.1.1 Normal requirements

- A free and opensource DNS server must be chosen.
- The chosen DNS server must be configured for local development as a Docker container
- The chosen DNS server must be able to run in OpenShift for production.
- A client for communicating with the DNS server must be implemented.
- A handler for handling DNS actions and information retrieval must be implemented.
- An action for provisioning new DNS zones must be implemented.
- The handler must be able to create new DNS records under a zone.
- The handler must be able to edit existing DNS records under a zone.
- The handler must be able to delete existing DNS records under a zone.
- The user must be able to create new DNS zones under an existing service.
- The user must be able to view DNS zones that are included in a service.
- The user must be able to create DNS records under a zone.
- The user must be able to edit DNS records under a zone.
- The user must be able to delete DNS records under a zone.

- New functionality must be tested via automated tests.

3.1.2 Expected requirements

- The user interface should have a good UX.
- The user interface should have a modern UI.
- Administrators should be able to create new services from existing resources.
- Administrators should be able to manage all resources that the existing and new handler support.

3.1.3 Extra requirements

- A way of logging actions and the status of services may be implemented on the backend.
- Users may be able to list individual service resources.
- Users may be able to view individual service resources.
- Users may be able to execute actions on individual service resources.

3.2 Functionality

This section shall provide a general overview of the new functionality provided for the users and administrators in the application as well as present descriptions on each individual function.

Figure 1 presents the new functionality on the web application that a user and administration will have access to after this project.

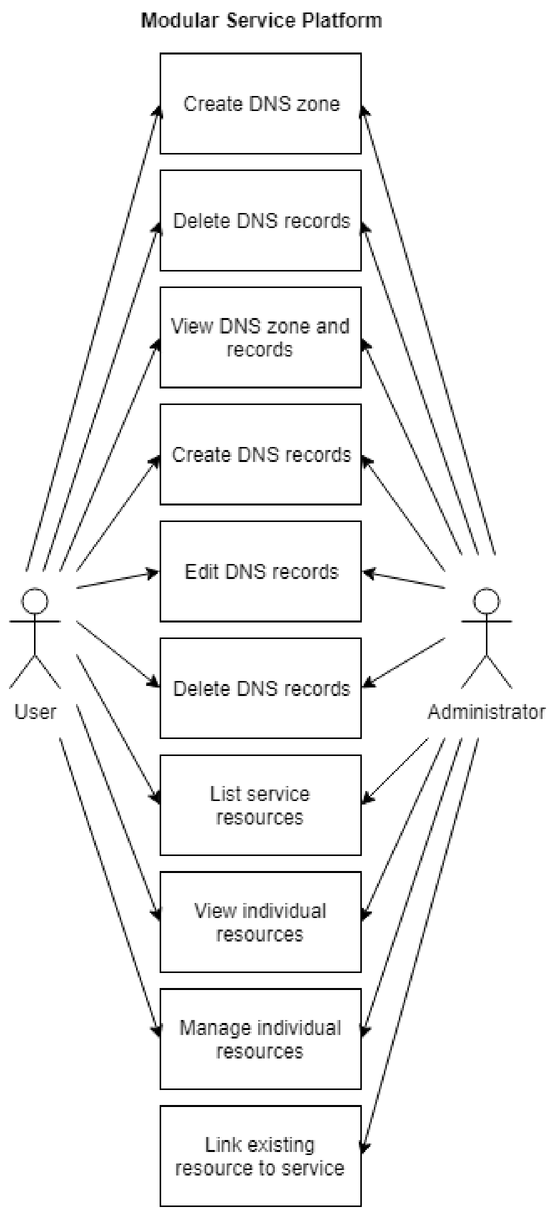


Figure 1. Main functionality provided by this project.

As seen in Figure 1, the first functionality is the creation of DNS zones. This means that the handler must support an action meant for provisioning a new service resource. Provided in Table 1 is the functional specification for this functionality.

Table 1. Functional specification of DNS zone creation

Case	DNS zone creation
Preconditions	Viewing an existing service with a DNS reservation
Input	The zone to create (e.g. puv.fi)
Description	The user presses the activate button on a DNS zone reservation, inputs the zone and clicks the activate button in the modal
Exceptions and errors	Invalid zone
Results and outputs	If the zone is valid, the service view should update and show the new resource instead of the reservation. A valid zone should be created in the DNS server.

The functionality starts with the user viewing a service that includes a DNS reservation. The user must click the “activate” button for the resource in question and input the zone to create. After a successful creation the service view will update with the new resource and its data displayed to the user. The design of this functionality can be seen in Figure 2.

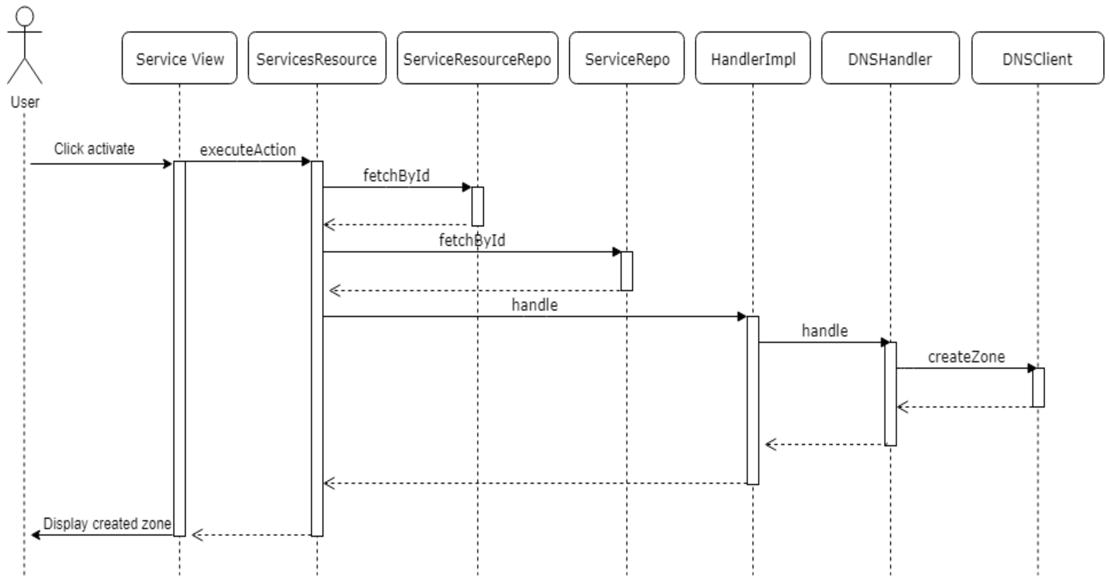


Figure 2. Implementation of the DNS zone creation functionality

The next functionality is for viewing a DNS zone's records. These records are displayed on the service view. And requires no other action from the user, except for viewing an existing service. The functional specification for this functionality is listed in Table 2.

Table 2. Functional specification for viewing a DNS zone and its records

Case	Display DNS zone and records
Preconditions	Existing DNS zone
Input	Selected service or zone.
Description	The user must select a service to view. The DNS zone data will be displayed to the user.
Exceptions and errors	No DNS zone linked as service re-source.
Results and outputs	DNS zone and records.

This functionality is essential for all other functionality relating to the DNS zone. If the user cannot see their DNS records, there is not much the that can be done. Figure 3 describes the implementation of this functionality.

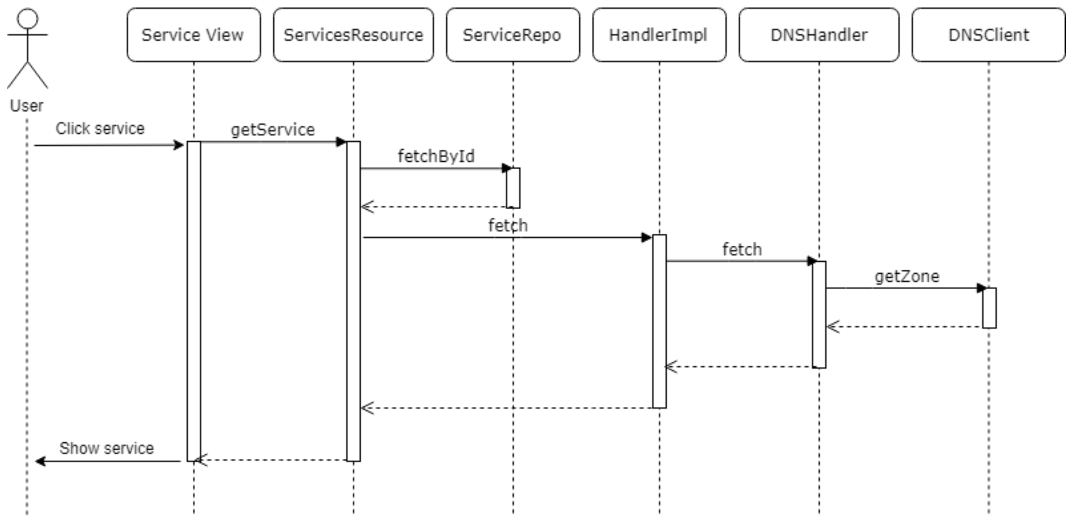


Figure 3. Implementation of the DNS zone viewing functionality

Once viewing has been implemented, the creation, editing and deletion of zones may be completed. All three of these functionalities will work the same way. Below is the functional specification for these in Table 3.

Table 3. Functional specification for DNS zone record editing.

Case	Creating, editing or deleting DNS records
Preconditions	Existing DNS zone.
Input	An action with a list of existing DNS records after the change.
Description	The user may edit the records in a zone by adding, editing or deleting as many records as needed. All changes are sent to the backend with one action and then compared to the existing set of records. Changes are updated to the DNS server as needed.
Exceptions and errors	Invalid DNS record or zone.
Results and output	The resulting updated DNS zone.

This functionality is the main aim of this project. Users should be able to make changes to their zone as they may change with time. Instead of executing an action per individual change. They can all be executed with one. Allowing multiple changes to be saved with one button press. Figure 4 describes the flow of events.

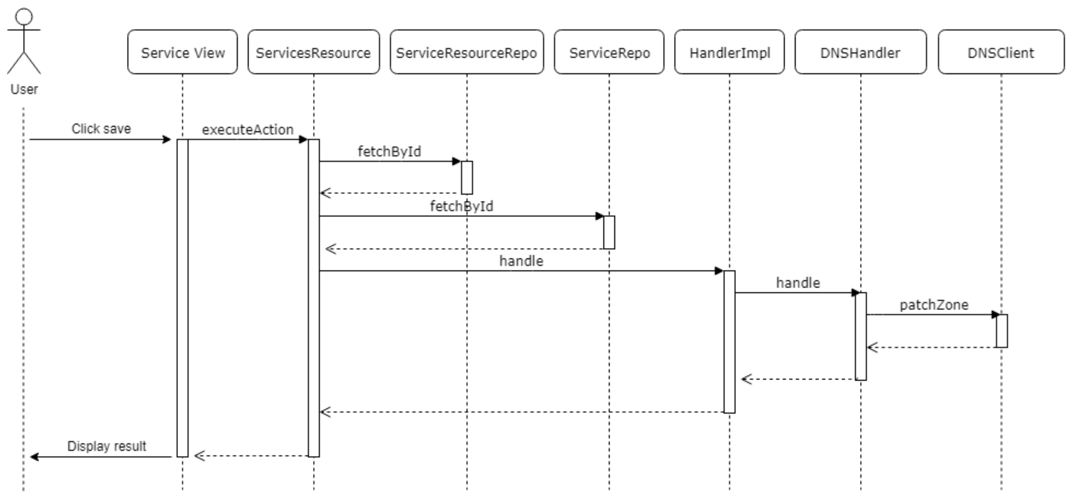


Figure 4. Implementation of DNS record editing functionalities

While the previous functionality only affects DNS, the following improve the usability of MSP in regards of other service resource types too. While it was meant that a service resource depends on a service. There are cases when it should not need to depend. Therefore, a requirement for this project is to implement a way of listing individual service resources. The functional specification for this functionality is presented in Table 4.

Table 4. Functional specification for listing service resources

Case	Listing service resources
Preconditions	Existing service resources
Input	Accessing the service resource list view
Description	Listing service resources with the ability to sort and search
Exceptions and errors	A resource provider may not respond.
Results and output	List of service resources

This functionality, skips the service querying and directly queries the service resource table directly. Below, the implementation for this functionality is presented in Figure 5.

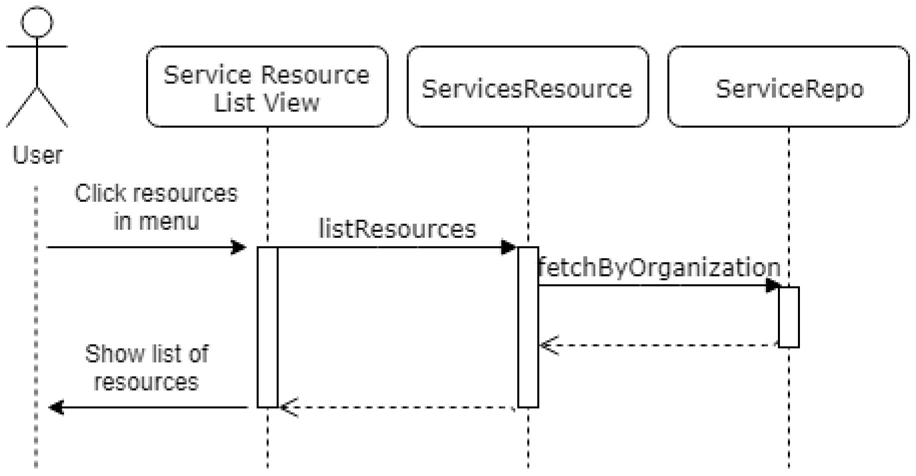


Figure 5. Implementation of listing service resources

From the service resource list, a user can select an individual service resource to view by clicking the desired resource. This will then draw the same service resource component that is used in the service view but only for the single component, by reusing the React component. The only difference is that a different endpoint must be created as this does not rely on a service id for fetching the service resource. Below, in Table 5, is the functional specification for this feature.

Table 5. Functional specification for viewing an individual service resource.

Case	Viewing an individual service resource.
Preconditions	An existing service resource.
Input	Clicking a service resource in the list view.
Description	View a single service resource the same way that a service may be viewed.
Exceptions and errors	A resource provider may not respond.
Results and output	The service resource and it’s data.

This will work by querying the service resource table for a certain resource and then using the right handler to fetch the data about the resource. In Figure 6 is the implementation for this functionality. The naming of “getData” from Handler to Client depends on the implementation, but the idea is similar in all of them.

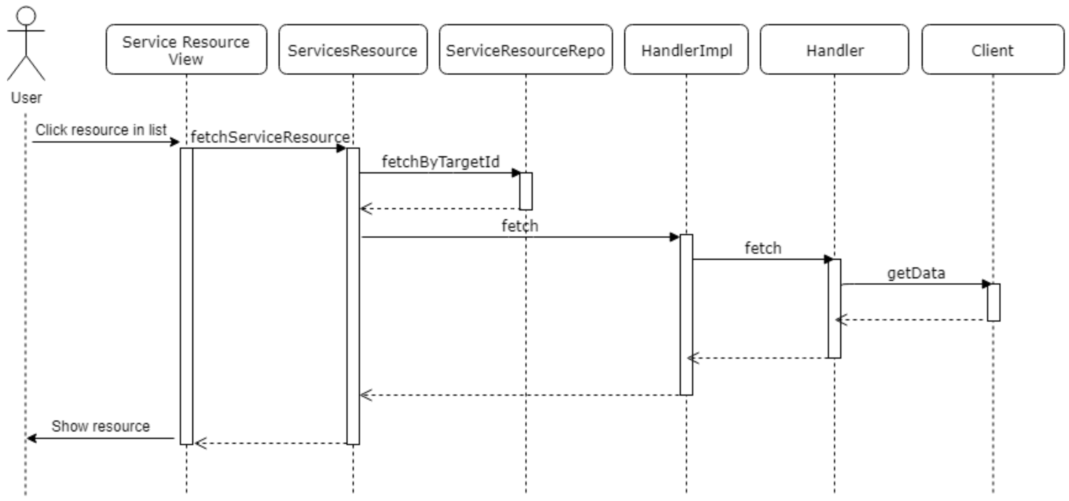


Figure 6. Implementation of viewing an individual service resource.

The next functionality is managing these individual service resource. This means that there must be a possibility to execute actions on the resource the same way that a user can execute actions in the service view. Again, a new endpoint is needed for this, while the frontend component may be reused quite well. The functional specification for this functionality is in Table 6.

Table 6. Functional specification for managing an individual service resource.

Case	Managing an individual service resource.
Preconditions	An existing service resource and the related actions defined.
Input	User action, this is usually via clicking a button in the frontend.
Description	The user executes an action on a service resource and the defined functionality must happen. This is the same as with a service, but from the context of a service resource.
Exceptions and errors	The resource provider may not respond.
Results and output	The updated service resource reflecting the changes.

The main task with this functionality is the new endpoint and connecting all the handlers to work with individual components. Figure 7 describes the implementation for this functionality.

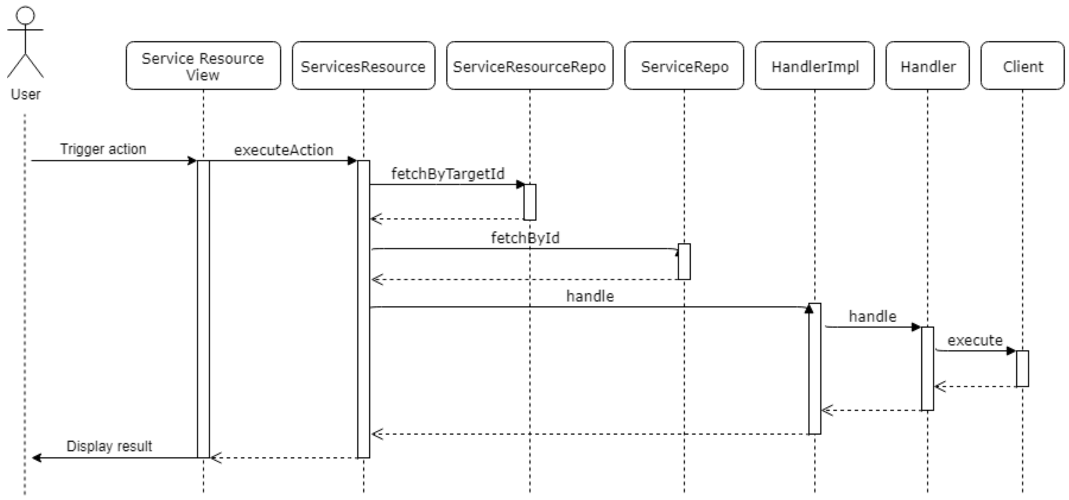


Figure 7. Implementation of managing an individual service resource

The last functionality is linking an existing service resource to a service. This allows for importing existing resources to MSP and letting customers manage their services without giving access to whole internal systems. The functional specification for this feature is presented in the Table 7.

Table 7. Functional specification for linking a service resource to a service.

Case	Link existing service resource to a service.
Preconditions	Existing service resource.
Input	User action, user must view the individual service resource and click the "Link" button and inputting the new service name and selecting a service template.
Description	The user clicks the "Link" button and a modal is opened asking for a service name and template. A service is created with that name and template with the service resource "created" in MSP under that service.
Exceptions and errors	Invalid service name.
Results and output	A new service with the service resource under it.

The main task with this functionality is creating an endpoint which unlinks the service resource from existing services, creates a new service and then adds a record in the database with a relation to the new service. Figure 8 describes the implementation for this functionality.

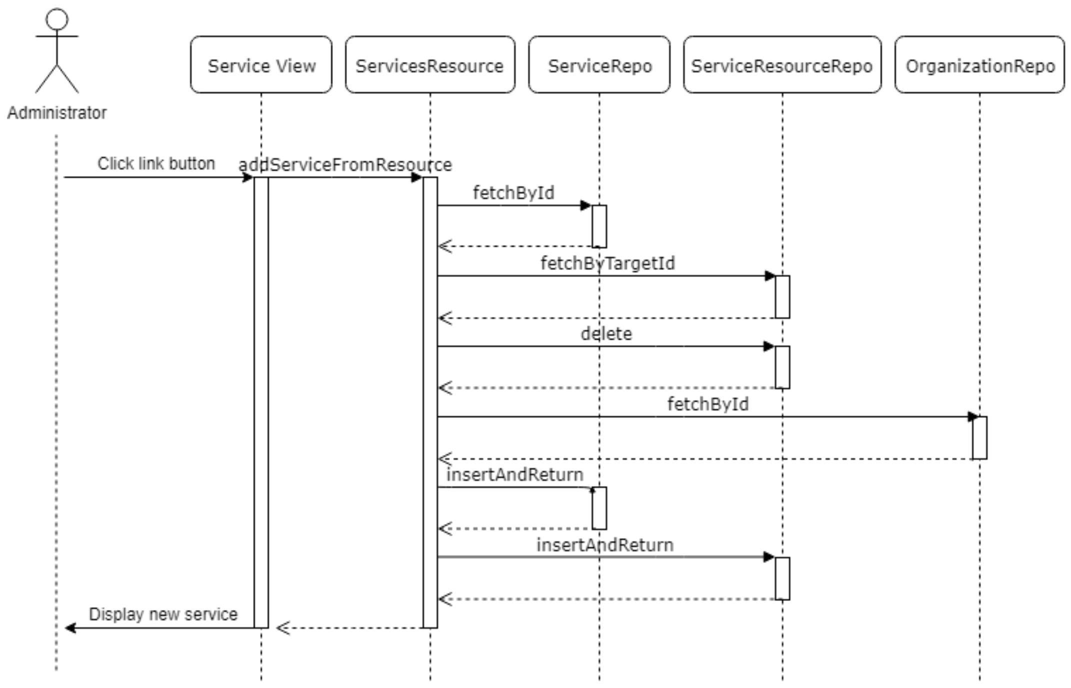


Figure 8. Implementation of linking an existing resource to a service

4 DESIGN

Designing the application is an important part of the process of software development and affects the result. Good design makes a good implementation possible. In this project the design consisted of selecting the DNS server used, designing the user interface as well as designing a new table for the database.

4.1 DNS Server

One of the major points of this project is to choose an authoritative DNS server which will persist the DNS records and serve them to the public. A major requirement is that the DNS server is free and open source. While most of this project is generic and will work with whatever server is chosen. The DNS server and its client in the backend must be specifically chosen. The two major options are BIND and PowerDNS. This chapter will compare the two and a decision will be made on which is more suitable for the project.

BIND is one of the most used DNS servers in the world. Development started in the early 1980s at the University of California and BIND stands for Berkeley Internet Name Domain. It is licensed under the permissive ISC license. /20/

A major strength of BIND is that it is very widely used and has been developed for a long time resulting in a robust server. Management of BIND is very simple as all zones and settings reside in text files. Which could be used with a VCS, such as Git or Subversion.

However, managing everything with text files may also be cumbersome when we want to programmatically perform modifications and there is no API included in BIND. We would need to update these text files and issue a reload in BIND. This would involve running shell commands from a different Docker container which is not the perfect solution.

PowerDNS is an open source DNS server, which was launched in 1999 and open sourced in 2002. PowerDNS is developed and supported by Open Xchange. It is licensed as GPL, which is another permissive open source license. PowerDNS is the only server which supports persisting records in most major database vendors, such as MySQL, PostgreSQL and Microsoft SQL Server. PowerDNS also allows scripting of DNS replies allowing dynamic records and filtering as well as provides an API for managing DNS records. /21/

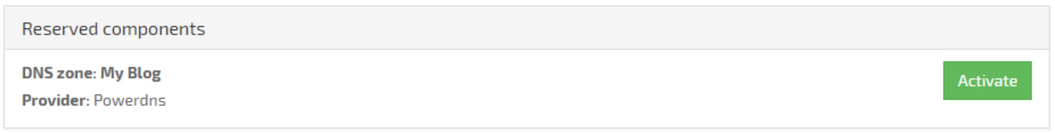
There are multiple strengths for PowerDNS. Being backed by a company that provides support makes it a good fit as if something goes wrong, paid support is available. However, it is not needed and PowerDNS is free to use. Another major strength is the programmability of the server. We can just write a client to interface with the DNS server over a REST API. However, PowerDNS is not as known and used as BIND. It is a much younger project.

Considering the merits of both DNS servers, PowerDNS seems to be the best fit as easy and reliable automation is key. Abstracting the storing of records in to a real database seems interesting and may result in better performance as well as an API allowing us to easy communication between Docker containers is the major selling point for this project.

4.2 User Interface

As external users are the main target audience of this application; a user-friendly and visually pleasing interface is a very important factor in determining the success of this project. The design of the frontend mainly consists of using React Bootstrap components, with some custom CSS styles. Bootstrap allows a developer to easily create modern and responsive websites, while React Bootstrap provides easy to use components. React also improves reusability of custom components.

With the main aim of this project being the DNS component, the very first component of the user interface is the creation of a service. While there is a create button for a reserved DNS component as seen in Figure 9 below, nothing happens when it is clicked.


Reserved components

DNS zone: My Blog

Provider: Powerdns

Figure 9. Reserved DNS component

When the user clicks the “Activate” button, a modal should appear with a form with fields for each piece of information that is needed. This modal must be reusable so that it may be used for other types of resources in the future. The way that this will work is that the “Activate”-action provides “Form Fields” which are used to dynamically create inputs for each form field and may be filled with a default value. Figure 10 below shows the activation modal. The DNS zone creation action only needs one field, which is the zone that is wanted.

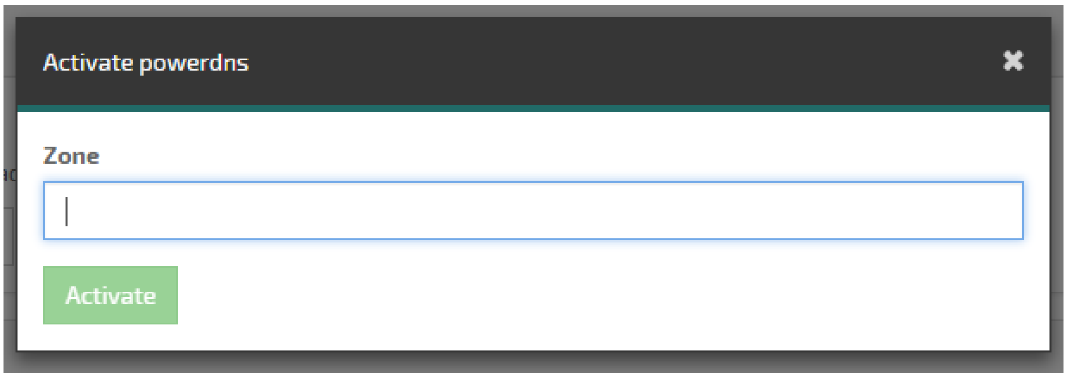


Figure 10. DNS activation modal

As seen in the Figure 10, with the zone input being empty, the activate button is disabled. As seen in Figure 11 below, the button becomes enabled when the input has been filled with the appropriate information.

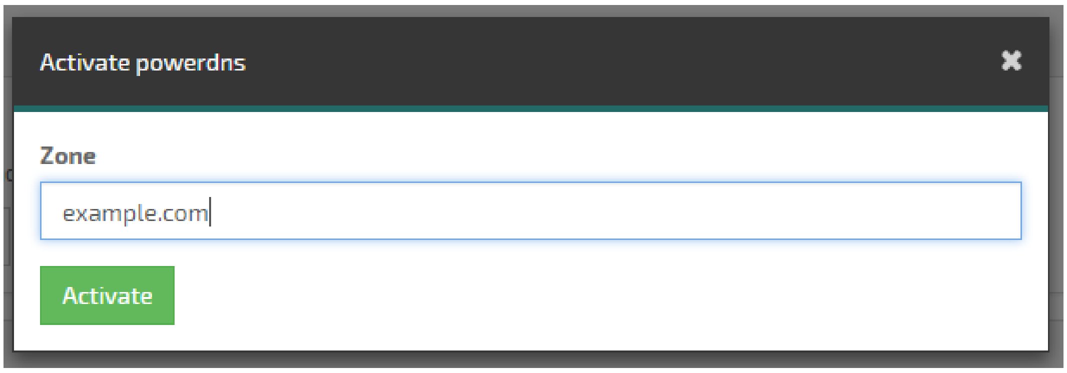


Figure 11. DNS activation modal with a zone

With DNS zones consisting of simple records, a table is a proficient way of displaying data to the user. For this reason, the React Bootstrap table component was chosen for displaying the records. In Figure 12 below, an empty DNS resource component is displayed after the zone was activated.

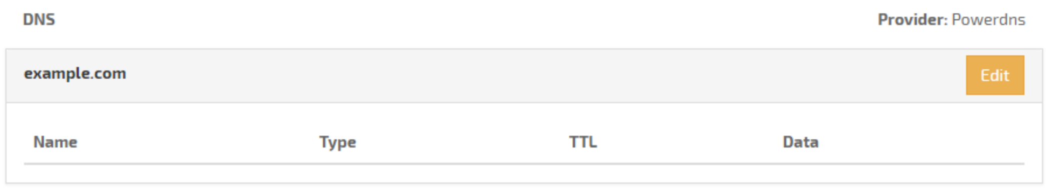


Figure 12. Empty DNS zone component

As may be seen, the DNS record only consists of 4 different values, which are all presented in the table. In the figure, the zone has just been created, therefore it is empty. There must be a way to add new records for the zone to be useful. The “Edit” button opens a modal, in which records can be added, edited or deleted as needed. Figure 13 displays the newly opened modal.

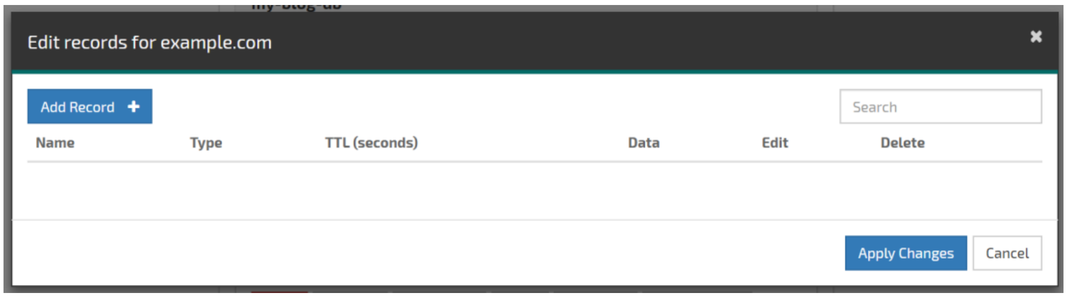


Figure 13. Empty DNS zone edit modal

Clicking the “Add Record” button adds a new row in the table with inputs for the values. Once the fields have been filled, the user can click the “Save” button or at any point the adding may be canceled. This may be observed in Figure 14 below.

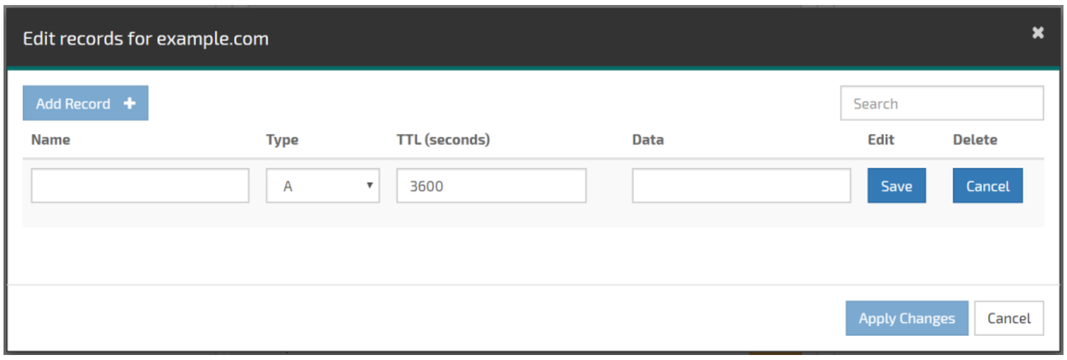
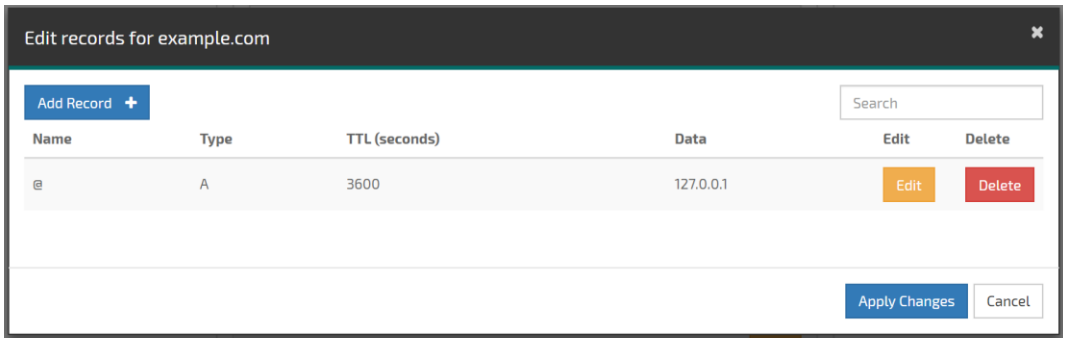


Figure 14. Adding a new record in the edit modal

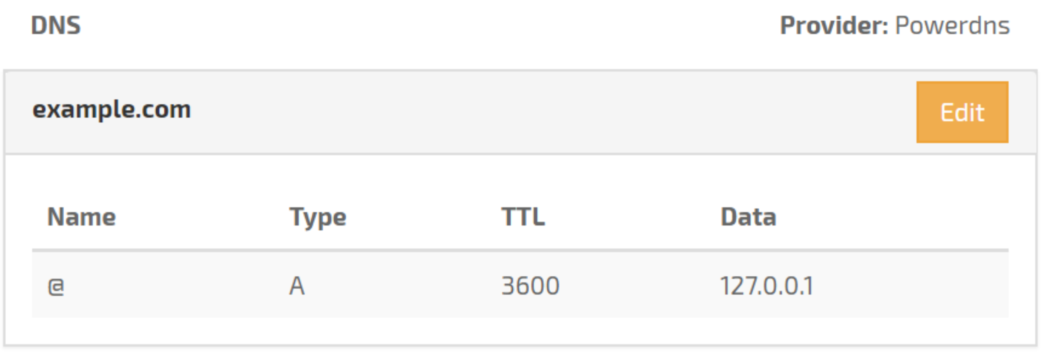
Once the save “Save” button is clicked the record will be displayed as a normal row with “Edit” and “Delete” buttons as seen in Figure 15.



Name	Type	TTL (seconds)	Data	Edit	Delete
@	A	3600	127.0.0.1	Edit	Delete

Figure 15. Saving a new DNS record

After editing and adding all the records the user may click the “Apply Changes” button which will persist the records in the DNS server. The reason for this is that we want to save all the records with one request as well as providing the user a step at which to confirm that they have made the right changes before updating critical data. A broken DNS record may affect their customers which may not be able to access their website. After the changes are applied, the edit modal is closed and the read-only view is updated to reflect the changes in the DNS provider as seen in Figure 16.



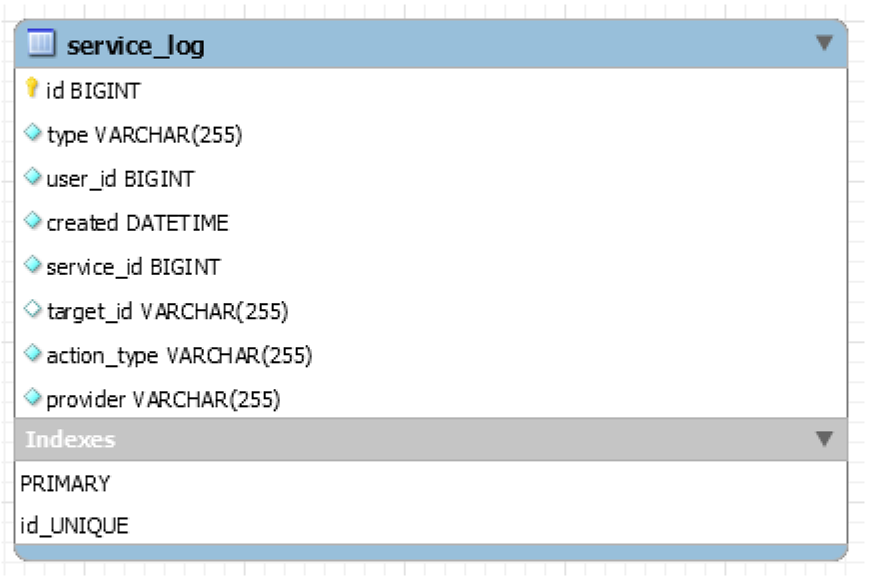
Name	Type	TTL	Data
@	A	3600	127.0.0.1

Figure 16. Read-only view showing the state of DNS

While the frontend seems simple, it provides a good user experience and looks very modern. It allows users to easily manage their DNS records, while providing steps to make sure that the changes are correct and cleanly presents the crucial data in a clean way.

4.3 Database

Having this project building on a previous project and most of the new data being handled by PowerDNS, the database was already designed. However, to implement service logging, a new table must be added for this functionality. The table must hold information about which service was affected, what was done, when was it done, by who and if it is an action of status change. The table was designed using MySQL Workbench and the table may be observed in Figure 17.



```
service_log
id BIGINT
type VARCHAR(255)
user_id BIGINT
created DATETIME
service_id BIGINT
target_id VARCHAR(255)
action_type VARCHAR(255)
provider VARCHAR(255)
Indexes
PRIMARY
id_UNIQUE
```

Figure 17. Service Log table design

As seen, “id” is the primary key of this table. It is auto incremented and is only used for selecting a certain row in the table. The column “type” is used to define if the logged row is an action or a status change as we want to log both in the same table. A status change is always the result of an action. However, not all actions trigger status changes. Actions are logged only for determining what happened and when. Status changes will be used in the future as billing data to determine the amount that a service will be billed for.

The column “user_id” is a foreign key, which points to a record in the “portal_user” table, which then points to a user in LDAP. This column is used to determine the person, that executed an action. The “created” column is a timestamp which will always be the datetime at when the record is inserted.

The column “service_id” is another foreign key, which points to a record in the “service” table, the service affected. The “target_id” is a nullable string representing an id in an external system. For example, in the case of DNS, the target_id would be the zone. This field is nullable because a status change affects the whole service, not a service resource. Together, the “target_id” and “provider” create a kind of “composite key” to a foreign system, which determines the exact service resource that an action was executed on. The “provider” could, for example, be “powerdns”, “openshift” or “redmine”. This means that for example if the “target_id” is “example.com” and the “provider” is “powerdns”, we immediately know that the “example.com” zone in PowerDNS was updated.

The “action_type” column may contain two types of data. If the log row is an action log, it contains the action that was executed. If it a status update, it contains the mode that the service is changed to. This could be “PROD”, for production, “DEV”, for development or “DELETED”, if the service is deleted.

5 IMPLEMENTATION

The Modular Service Platform consists of two subprojects. The backend, written in Java; and the frontend, which is a SPA using React. These two components communicate between each other utilizing REST over HTTP. The backend serves pre-determined endpoints which either output data or expect data to be inputted, which is then processed, acted upon and persisted in to a database. The frontend is used to present this data as well as input new data and execute actions.

5.1 Backend

A major component of the Java application is Dropwizard, a framework for writing REST services. It contains the business logic of the application and interfaces with external systems, such as OpenShift, Redmine, MariaDB and after this project, PowerDNS. The backend also contains all migrations that are performed on the database using Liquibase. The most important packages in the codebase are “repo”, “resources”, “core”, “handlers” and “api”. Most work was performed on the “handlers” package as that is where the clients and logic for communicating with external systems, in this case PowerDNS, reside.

5.1.1 PowerDNS client

While PowerDNS provides a REST API, there was no suitable client for Java. A simple client which utilizes okhttp3 for managing and creating zones was implemented. The PowerDNS API has token-based authentication which had to be taken in to consideration, as well as the zone data had to be modeled as Java classes. The following code was implemented in a “powerdns” package under the “handlers” package.

The first task was to model the DNS zone data as POJOs with the needed getters and setters, which then would be serialized and deserialized to and from JSON strings using Jackson. These classes are quite straightforward as can be seen in Code Snippet 6.

```
// Annotation for ignoring unknown properties. This might be needed if PowerDNS decides
to add more properties in the future and our POJO should still work.
@JsonIgnoreProperties(ignoreUnknown = true)
public class PowerDnsZone {
    @JsonProperty
    private String name;

    @JsonProperty
    private List<PowerDnsRecordSet> rrsets;

    public PowerDnsZone() {}

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public List<PowerDnsRecordSet> getRrsets() {
        return rrsets;
    }

    public void setRrsets(List<PowerDnsRecordSet> rrsets) {
        this.rrsets = rrsets;
    }
}
```

Code Snippet 6. POJO with Jackson annotations

After modelling the needed POJOs, the PowerDNS client itself could be implemented. There are 5 notable methods in the client, which are used for fetching, creating, updating and deleting zones. There is a method for fetching a single zone, as well as a method for fetching all zones. The first method which was implemented was the creation of a DNS zone and it may be seen in Code Snippet 7 below.

```
public Optional<ServiceResource> createZone(
        fi.jubic.serviceplatform.core.Service service, String zone
) {
        // Filter PowerDNS resource from service which does not have a target id.
        ServiceResource sr = service.components()
                .stream()
                .filter(s -> s.provider()
                        .equals(Provider.POWERDNS) && s.targetId() == null
```

```
            ).findFirst()
            .orElseThrow(NotFoundException::new);

        // Construct a new POJO with the zone name for sending to PowerDNS API
        PowerDnsCreateZone powerDnsCreateZone = new PowerDnsCreateZone(
            zone + "."
        );

        // Get default nameservers from configuration (ns1.example.com|ns2.example.com)
        List<String> nameServers = Arrays.stream(configuration.getNameservers()
            .split("\\|"))
            .map(n -> n + ".")
            .collect(Collectors.toList());
        powerDnsCreateZone.setNameservers(nameServers);

        try {
          // Create request body via deserializing the POJO and set mediatype to JSON.
          RequestBody body = RequestBody.create(
              MediaType.parse("application/json, charset=utf-8"),
              mapper.writeValueAsString(powerDnsCreateZone)
          );

          // Build request and set authorization token from configuration.
          Request request = new Request.Builder()
              .url(configuration.getUrl())
              .header("X-API-Key", configuration.getToken())
              .post(body)
              .build();
          // Send request to PowerDNS API
          Response response = client.newCall(request).execute();

          // If response code does not start with a 2 (e.g. 200) return an empty Optional
          if (!response.isSuccessful()) {
            return Optional.empty();
          }

          // Set the service resource to activated and set new target id
          sr = sr.toBuilder()
              .setActivated(true)
              .setTargetId(zone)
              .build();
          // Update the service resource information in the database.
          serviceResourceRepo.update(sr);
          // Return an Optional of the updated service resource.
          return Optional.of(sr);
        } catch (IOException e) {
          e.printStackTrace();
        }
        return Optional.empty();
    }
```

Code Snippet 7. PowerDNS client method for creating zone

As seen in Code Snippet 7, to create a new zone in PowerDNS, from the user's point of view, the only variable needed is the zone name, for example "example.com”. From the developer’s point of view, the nameservers also have to be set. The POJO is then deserialized in to the request body, which is built with a header containing the authorization token. This request is sent to PowerDNS and if the creation was successful, the service resource is set to activated and the zone name is set as the target id. This is then updated in the database and the updated service resource is then returned. As one may observe, the return type is an Optional, which is a way to not have to return null values.

For fetching DNS zones, the request itself looks very similar, instead of a POST request with a body, a GET request with the target id in the URL is all that is needed. The deserialized POJO is then transformed in to the data model consumed by our application.

Deleting DNS zones is even more straightforward as instead of a GET request, a DELETE request is sent. The response then indicates if the DNS zone was deleted and a Boolean is returned indicating if the delete was successful or not. A service resource is not returned because the service resource was just deleted.

The most interesting part of the client is the way patching records works. A list of all records after the update is supplied to the method and the method uses the fetch zone method to get the records before the update. The two lists are compared to see which records need to be deleted, which records need to be updated and which records need to be created. The resulting list is sent to PowerDNS to be processed. This method may be seen in Code Snippet 8 below.

```
public Boolean patchRecord(List<UserData> userData, String zone) {
   // Fetch the zone currently in PowerDNS
   PowerDnsZone dnsZone = getZone(zone).orElseThrow(NotFoundException::new);
   List<PowerDnsRecordSet> records = dnsZone.getRrsets();
   // Compare the old and new zones for changes
   List<PowerDnsRecordSet> patch = compare(zone, records, userData);

   PowerDnsAddRecord addRecord = new PowerDnsAddRecord(patch);

   // Send changes to PowerDNS.
   try  {
      RequestBody body = RequestBody.create(
            MediaType.parse("application/json, charset=utf-8"),
                mapper.writeValueAsString(addRecord)
            );

            Request request = new Request.Builder()
                .url(configuration.getUrl() + "/" + zone)
                .header("X-API-Key", configuration.getToken())
                .patch(body)
                .build();

            Response response = client.newCall(request).execute();

            return response.isSuccessful();
        } catch (IOException e) {
            e.printStackTrace();
        }

        return false;
    }
```

Code Snippet 8. Updating multiple records with a single request

The way that this method works is that it first fetches the old record set from PowerDNS. It then called the compare method, which iterates over each of the records in the updated zone. The compare method may be observed in Code Snippet 9.

```
public List<PowerDnsRecordSet> compare(String zone, List<PowerDnsRecordSet> rec-
ords, List<UserData> userData) {
    List<PowerDnsRecordSet> result = new ArrayList<>();

    // Iterate over all the records sent by user.
    userData.forEach(u -> {
        Map<String, String> variables = u.getVariables();
        String type = variables.get("TYPE").toUpperCase();
        Long ttl = Long.parseLong(variables.get("TTL"));
        String content = variables.get("RECORD");
        String name = (variables.get("NAME") + "." + zone + ".")
            .replace("@.", "");

        if (type.equals("CNAME")) {
            content = content + ".";
        }

        // Get old record with same name and type if it exists.
        Optional<PowerDnsRecordSet> oldOptional = records.stream()
            .filter(r -> r.getName().equals(name) && r.getType().equals(type))
            .findFirst();
        Optional<PowerDnsRecordSet> newOptional = Optional.empty();

        // If no record existed, create a new record.
        if (!oldOptional.isPresent()) {
            newOptional = result.stream()
                .filter(r -> r.getName().equals(name) && r.getType().equals(type))
                .findFirst();
        }

        // Add change to changeset if the record is new or changed.
        if (oldOptional.isPresent() || newOptional.isPresent()) {
            PowerDnsRecordSet old;
            if (oldOptional.isPresent()) {
                old = oldOptional.get();
                records.remove(old);
                old.setRecords(Lists.newArrayList());
            } else {
                old = newOptional.get();
            }

            if (newOptional.isPresent()) {
                result.remove(old);
            }

            PowerDnsRecord newRecord = new PowerDnsRecord();
            newRecord.setContent(content);
            newRecord.setDisabled(false);

            List<PowerDnsRecord> recordList = old.getRecords();
            recordList.add(newRecord);

            old.setRecords(recordList);
            old.setTtl(ttl);
            old.setChangetype("REPLACE");

            result.add(old);
        } else { // Else keep old record.
            PowerDnsRecord record = new PowerDnsRecord();
            record.setDisabled(false);
            record.setContent(content);

            PowerDnsRecordSet rrset = new PowerDnsRecordSet();
            rrset.setRecords(Lists.newArrayList(record));
            rrset.setChangetype("REPLACE");
            rrset.setName(name);
            rrset.setType(type);
            rrset.setTtl(ttl);

            result.add(rrset);
        }
    });

    // Delete leftover records from old recordset.
    records.forEach(r -> {
        r.setRecords(Lists.newArrayList());
        r.setChangetype("DELETE");

        result.add(r);
    });
    return result;
}
```

Code Snippet 9. Comparing DNS records for changes

Code Snippet 9 demonstrates how the comparison iteration works. In this iteration it compares each new record to the old records. If a change is detected it is added to a patch list. After all the updated records have been iterated through, we have a list of records left over that were not in the list of updated records. These are marked for deletion and a request may be built with the patched and deleted records in the zone.

5.1.2 PowerDNS Handler

After the client has been implemented, the PowerDNS handler may be created. The role of the handler is to filter out actions intended for the handler which are then passed on to the client with the relevant data. The way that handlers work is that each incoming action is given to all handlers which then either act upon the action or return an empty Optional, which means that the handler does not support the action. If no handler supports and action, it is added as a Redmine ticket through the fallback Redmine handler so that it can be handled later by administrators manually. This allows us to add a way to complete actions that are not supported yet.

The handler consists of two Java classes and a shared handler interface. The first class is the configuration class, which defines all the configuration that is needed by a handler, such as API URLs and authentication tokens. The second class is the actual handler class which implements the handler interface, a contract for all handlers on which methods must be implemented.

The handler interface defines 6 methods, which are “register”, “componentOf”, “handle” (for resource templates), “handle” (for actions), “fetch” and “getComponents”. The register method is for registering websocket tunnels for external services, but in the case of PowerDNS, there are none, which means that this method just returns void.

The handle methods are the most interesting ones in the handler classes and are very similar. They utilize method overloading which means that while having the same name, they have different input parameters. One of them is mainly for creating new service resources from a resource template on service creation, while the other handles actions executed by a user. In Code Snippet 10 below, the resource template handle method is shown.

```
@Override
public Optional<ServiceComponentView> handle(
        Service service,
        ResourceTemplate resourceTemplate,
        List<UserData> userData,
        Long userId
) {
    // If the resource template is not for PowerDNS, return empty.
    if (!Objects.equals(resourceTemplate.provider(), Provider.POWERDNS)) {
        return Optional.empty();
    }

    switch (Objects.requireNonNull(resourceTemplate.type())) {
        case ActionType.POWERDNS_NEW_ZONE:
            // Get the provided user input for creating a new zone.
            Optional<UserData> data = userData.stream()
                .filter(ud -> Objects.equals(ud.getActionId(), resourceTemplate.id()))
                .findFirst();

            // If no user input provided, do not create a new zone.
            if (!data.isPresent()) {
                return Optional.empty();
            }

            Map<String, String> variables = data.get().getVariables();
            // Get zone name.
            String zone = variables.get("ZONE").toUpperCase();
            // Create new zone and return the data to user.
            return fetch(
                client.createZone(service, zone)
                .orElseThrow(BadRequestException::new)
            );

        // If no provided action is found, return empty.
        default:
            return Optional.empty();
    }
}
```

Code Snippet 10. PowerDNS handler resource template handle method

As seen in Code Snippet 10 above, the handle method is quite simple as most of the logic is in the client itself. The method checks that it is equipped to handle the resource template by comparing the provider. After this in the switch statement, the action type is checked. Depending on the action type, a new zone is created with the client and the fetch method is used to return the new DNS zone or an empty Optional is return to signal that the handler does not support this type of resource template.

The editing, adding and deleting actions are handled by the second handle method. These are actions which are executed on an existing service resource and therefore do not have anything to do with resource templates. Another task of this method is to log actions in to the service log. This method is demonstrated in Code Snippet 11 below.

```
    @Override
    public Optional<ServiceComponentView> handle(
            Service service,
            Action action,
            List<UserData> userData,
            Long userId
    ) {
        // If the requested action's provider is not PowerDNS, return empty.
        if (!action.provider().equals(Provider.POWERDNS)) {
            return Optional.empty();
        }

        ServiceResource sc = null;
        /* If the service is not known get component from the database, otherwise get com-
ponent from service */
        if (service == null && !action.type().equals(ActionType.POWERDNS_NEW_ZONE)) {
            sc = getComponents(false)
                    .stream()
                    .map(ServiceComponentView::toCore)
                    .filter(c -> Objects.equals(c.targetId(), action.targetId()))
                    .findFirst()
                    .orElseThrow(NotFoundException::new);
        } else if (!action.type().equals(ActionType.POWERDNS_NEW_ZONE)) {
            sc = service.components()
                    .stream()
                    .filter(c -> Objects.equals(c.targetId(), action.targetId()))
                    .findFirst()
                    .orElseThrow(NotFoundException::new)
                    .toBuilder()
                    .setService(service)
                    .build();
        }

        ServiceLog serviceLog;
```

```
switch (action.type()) {
    case ActionType.POWERDNS_NEW_ZONE:
        Map<String, String> variables = userData.get(0).getVariables();
        String zone = variables.get("ZONE");

        // Create new zone and fetch it from PowerDNS
        Optional<ServiceComponentView> result = fetch(
                client.createZone(service, zone)
                        .orElseThrow(BadRequestException::new)
        );

        if (result.isPresent()) {
            // Log the creation to database
            serviceLog = ServiceLog.builder()
                    .setType(ServiceLogType.STATUS)
                    .setUser(PortalUser.builder().setId(userId).build())
                    .setCreated(DateTime.now())
                    .setTargetId(result.get().toCore().targetId())
                    .setActionType(ServiceStatus.CREATED.name())
                    .setProvider(Provider.POWERDNS)
                    .setServiceResource(result.get().toCore())
                    .build();

            serviceLogRepo.insert(serviceLog);

            serviceLogRepo.insert(
                    serviceLog.toBuilder()
                            .setActionType(ServiceStatus.PROD.name())
                            .build()
            );
        }

        return result;

    case ActionType.POWERDNS_PATCH_RECORD:
        // Edit zone with records.
        if (client.patchRecord(userData, action.targetId())) {
            return fetch(sc);
        }

        return Optional.empty();

    case ActionType.DELETE:
        // Mark zone as deleted in database.
        serviceLog = ServiceLog.builder()
                .setUser(PortalUser.builder().setId(userId).build())
                .setProvider(Provider.POWERDNS)
                .setCreated(DateTime.now())
                .setType(ServiceLogType.STATUS)
                .setActionType(ServiceStatus.DELETED.name())
                .setTargetId(sc.targetId())
                .setServiceResource(sc)
                .build();

        serviceLogRepo.insert(serviceLog);

        return fetch(sc);
```

```
            default:
                return Optional.empty();
        }
    }
```

Code Snippet 11. PowerDNS handler action handle method

In Code Snippet 11, the handler supports three types of actions, creating new zones, patching records as well as deleting zones. The reason for this method also supporting the creation of new zones is that the DNS component may not be activated on service creation. Instead the user may later activate it by clicking the activate button in the frontend in an existing service.

Another observation may be made that the delete action does not actually do anything other than set the resource as deleted in the service log. This functionality is as requested, and it allows the customer to migrate the DNS with an administrator removing the zone later. The functionality for deleting a DNS record does exist in the client and may be used later if deemed necessary.

The fetch method is used to format the incoming data from the PowerDNS client into the data presented to the user. It is used in both handle methods to return the service resource after actions or resource templates have been executed. Code Snippet 12 demonstrates this method.

```
@Override
public Optional<ServiceComponentView> fetch(ServiceResource component) {
    if (!component.provider().equals(Provider.POWERDNS)) {
        return Optional.empty();
    }

    // If zone is not activated, return empty zone.
    if (!component.activated()) {
        return Optional.of(
                ServiceComponentView.builder()
                        .setServiceComponent(component)
                        .setUpdatable(updatableKinds.contains(component.kind()))
                        .setDeletable(deletableKinds.contains(component.kind()))
                        .setDisplayName(component.name())
                        .setActions(Lists.newArrayList(ActionView.of(Actions.newZone)))
                        .build()
        );
    }
```

```
switch (component.kind()) {
   case PowerDnsKind.ZONE:
      List<PowerDnsRecordResponse> records = new ArrayList<>();

      // Fetch zone from PowerDNS
      PowerDnsZone z = client.getZone(component.targetId())
            .orElseThrow(BadRequestException::new);

      if (z.getRrsets() != null) {
         // Format zone records for user interface.
         z.getRrsets().forEach(rs -> {
            if (rs.getType().equals("SOA") || rs.getType().equals("NS")) {
               return;
            }

            String name = rs.getName()
                  .replace(component.targetId() + ".", "");

            if (name.length() == 0) {
               name = "@";
            }

            if (name.length() > 1) {
               name = name.substring(0, name.length() - 1);
            }

            String finalName = name;

            rs.getRecords()
                  .forEach(r -> {
                     if (rs.getType().equals("CNAME")) {
                        r.setContent(r.getContent()
                        .substring(0, r.getContent().length() - 1));
                     }

                     records.add(new PowerDnsRecordResponse(
                           rs.getType(),
                           finalName,
                           rs.getTtl(),
                           r.getContent())
                     );
                  });
         });
      }

      Optional<PowerDnsZoneResponse> zone = Optional.of(
         new PowerDnsZoneResponse(component.targetId(), records)
      );

      List<Action> actions = new ArrayList<>();

      // Add zone records editing as a supported action.
      actions.add(Actions.patchRecord
            .toBuilder()
            .setTargetId(component.targetId())
            .build());

      return Optional.of(
            ServiceComponentView.builder()
```

```
                            .setServiceComponent(component)
                            .setDisplayName(component.name())
                            .setUpdatable(updatableKinds.contains(component.kind()))
                            .setDeletable(deletableKinds.contains(component.kind()))
                            .setActions(
                                actions.stream()
                                    .map(ActionView::of)
                                    .collect(Collectors.toList())
                            ).setPowerdnsZone(zone.orElse(null))
                            .build()
            );
        default:
            return Optional.empty();
    }
}
```

Code Snippet 12. PowerDNS handler fetch method

The task of this method is to return data which is used to present information about the service resource in the frontend. If a service resource is not activated, the resource will be presented with a button which allows the user to activate it. Otherwise, the record data is cleaned up to be more user friendly by using @ to represent the root zone in data fields as well as removing the last period from domains as users are not used to it being there even though it is a part of the specification for DNS.

Another task of the method is to define the actions that the service resource supports. These are defined elsewhere, but the actions for the PowerDNS handler are listed in the action handle method. This method also may return an empty Optional which means that the handler does not support this type of service resource.

The “getComponents” method is related to the service resource list requirement. The task of this method is to gather all the service resources that the provider knows of. This method may be seen in Code Snippet 13.

```
@Override
public List<ServiceComponentView> getComponents(Boolean linked) {
    // Get all zones from powerdns
    return client.getZones()
            .stream()
            // Filter all existing linked zones if the linked parameter is false.
            .filter(z -> !linked ||
                !serviceResourceRepo.fetchByTargetId(z.getName()).isPresent())
                .map(z -> ServiceResource.builder()
                        .setTargetId(z.getName().substring(0, z.getName().length() - 1))
                        .setName(z.getName().substring(0, z.getName().length() - 1))
                        .setKind(PowerDnsKind.ZONE)
                        .setProvider(Provider.POWERDNS)
                        .build()
                ).map(sr -> ServiceComponentView.builder()
                        .setServiceComponent(sr)
                        .build()
                ).collect(Collectors.toList());
    }
}
```

Code Snippet 13. PowerDNS handler getComponents method

This method supports filtering out already linked resources from the list. This means that a has been linked to the service resource and only unlinked resources are returned. In the case of PowerDNS it utilizes the “getZones” method of the PowerDNS client.

5.1.3 Resource List

The resource list functionality depends on two different components. The first change is to the ServicesResource class. This class defines the different HTTP endpoints available. An endpoint for listing the resources under and organization was added, and it may be observed in Code Snippet 14.

```
@GET
@Path("/resources")
@ApiOperation("List service resources")
public List<ServiceComponentView> listResources(
  @HeaderParam("X-Portal-User") LongParam userId,
  @HeaderParam("X-Portal-User-Organization") LongParam organizationId
) {
  // Fetch all service resources under organization.

  return serviceResourceRepo.fetchByOrganization(organizationId.get())
    .stream()
    // Only show activated resources
    .filter(ServiceResource::activated)
    .map(s -> ServiceComponentView.builder()
        .setServiceComponent(s)
        .build()
    ).collect(Collectors.toList());
}
```

Code Snippet 14. ServicesResource listResources method

This method is an example of using the Streams API, which was introduced in Java 8. A list of service resources is fetched from the database with the provided organization id and turned in to a stream. This stream is then filtered for only activated resources. The map method then transforms the service resource objects in to ServiceComponentView objects. These are what are served as JSON to the frontend via REST. At the end the stream is collected, which means that it is turned back in to a list. Streams are a good introduction to Java and make methods shorted, less verbose while still maintaining readability.

The second method which needed to be implemented is the fetchByOrganization in the ServiceResourceRepo. This method is called from the listResources method. The method and its public overload may be observed in Code Snippet 15.

```
    public List<ServiceResource> fetchByOrganization(Long organization) {
        return fetchByOrganization(organization, false);
    }

    private List<ServiceResource> fetchByOrganization(Long organization, Boolean delet-
ed) {
        /* Select services resources from database which have the provided organization id
and the target id is not null. Join the service table on the service id value as well as organ-
ization table on the organization id. */
        return DSL.using(this.configuration)
                .select()
                .from(SERVICE_RESOURCE)
                .leftJoin(SERVICE)
                .on(SERVICE.ID.equal(SERVICE_RESOURCE.SERVICE_ID))
                .leftJoin(ORGANIZATION)
                .on(ORGANIZATION.ID.equal(SERVICE_RESOURCE.ORGANIZATION_ID))
                .where(
                        ORGANIZATION.ID.eq(organization)
                                .and(SERVICE_RESOURCE.TARGET_ID.isNotNull())
                ).fetch()
                .stream()
                .map(ServiceResourceRepo::map)
                .filter(Optional::isPresent)
                .map(Optional::get)
```

```
                    .filter(sr -> deleted || sr.deleted() == null)
                    .collect(Collectors.toList());
        }
```

Code Snippet 15. ServiceResourceRepo fetchByOrganization method

Code Snippet 15 demonstrates how method overloading with different parameters works in Java. The actual logic and more refined parameters is hidden from other classes while an abstracted method which ignores deleted service resources is provided for other classes to use. As the two methods have different parameters, they may use the same method name and return type.

The fetchByOrganization method uses jOOQ and its SQL-like chainable method syntax to query the database for the wanted records. A developer that is familiar with SQL may easily understand and use jOOQ to abstract away the database handler and generate corresponding POJO classes from the database schema.

In this instance, we are selecting all from the Service Resource table and joining the accompanying service and organization information where the organization id is the same as the provided parameter. The resulting records are fetched in to a list of “ServiceResourceRecord” objects. Utilizing the Stream API in Java, these records are mapped in to ServiceResource objects and empty Optionals are filtered out. A “ServiceResourceRecord” may become an empty Optional if, for instance, the ID column in the database is null for a record.

5.1.4 Service Logger

The final functionality in the backend is the service status logging. This was implemented by creating a new “ServiceLogRepo” for writing log rows in to the database. This class only has one method, which is for inserting the logs and may be observed in Code Snippet 16.

```
public Boolean insert(ServiceLog serviceLog) {
    // Insert servicelog in to database and return true if successful.
    return DSL.using(this.configuration)
        .insertInto(SERVICE_LOG)
        .columns(
            SERVICE_LOG.ACTION_TYPE,
            SERVICE_LOG.CREATED,
            SERVICE_LOG.PROVIDER,
            SERVICE_LOG.RESOURCE_ID,
            SERVICE_LOG.TYPE,
            SERVICE_LOG.USER_ID,
            SERVICE_LOG.TARGET_ID
        ).values(
            serviceLog.actionType(),
            new Timestamp(System.currentTimeMillis()),
            serviceLog.provider(),
            serviceLog.serviceResource().id(),
            serviceLog.type().toString(),
            serviceLog.user().id(),
            serviceLog.targetId()
        ).execute() == 1;
}
```

Code Snippet 16. ServiceLogRepo insert method

In Code Snippet 16, the method takes a ServiceLog object and maps the object to the database columns and behind the scenes creates a SQL query. This query is executed and if the return value is 1 the method will return true, otherwise it will return false, which means that the insert did not go through.

This insert method is called from the handlers when an action which modified the status of a service is executed. An example of this is the deletion of a PowerDNS zone. The creation of a ServiceLog object may be observed in Code Snippet 5 under the “case ActionType.DELETE”.

5.2 Frontend

The frontend is what the end user sees in their browser. In this case, the frontend uses React for developing a SPA. Various build tools, such as Babel are used to enable us to write modern, ECMAScript 6, JavaScript, while our codebase still working with older browsers. Babel also allows us to use JSX, which is mixture of JavaScript, with HTML-like syntax additions. We basically create “new” HTML tags, which consists of children, normal HTML tags, and functionality, JavaScript. These tags are called components.

5.2.1 PowerDNS

The PowerDNS component consists of two subcomponents, the read-only table and the edit modal. The role of the table is display the current data in PowerDNS for a zone, while the edit modal allows the user to edit a data and submit it to PowerDNS through MSP. The component for the edit modal may be seen in Code Snippet 17 below.

```
function ModalContent ({
  powerDnsZoneComponent,
  actions
}: ModalContentProps) {
  // Get needed variables from component.
  let {
    sortedRecords,
    edit,
    search,
    pagination
  } = powerDnsZoneComponent

  // Only get the records that should be showed according to pagination.
  let records = sortedRecords
    .skip(pagination.page * pagination.limit)
    .take(pagination.limit)
  const paginationItemsCount =
    Math.ceil(sortedRecords.size / pagination.limit)

  return (
      <Table striped responsive>
        <thead>
          <tr>
            <th>Name</th>
            <th>Type</th>
            <th>TTL (seconds)</th>
            <th>Data</th>
            <th>Edit</th>
            <th>Delete</th>
          </tr>
        </thead>

        <tbody>
          {
            // Create rows for each record.
            records.map((p, i) => edit.editOn && edit.index === i
              ? (<EditRow
                key={i}
                name={edit.name}
                type={edit.type}
                ttl={edit.ttl}
                record={edit.record}
                onNameEdit={actions.powerDnsZoneComponent.editName}
                onTypeEdit={actions.powerDnsZoneComponent.editType}
                onRecordEdit={actions.powerDnsZoneComponent.editRecord}
                onTtlEdit={actions.powerDnsZoneComponent.editTtl}
                onCancel={actions.powerDnsZoneComponent.stopRecordEdit}
                onSave={() => {
                  actions.powerDnsZoneComponent.saveRecord()
                  actions.powerDnsZoneComponent.stopRecordEdit()
                }} />
              )
              : (
               <tr key={i}>
                <td style={{ paddingTop: '14px' }}>{p.name}</td>
                <td style={{ paddingTop: '14px' }}>{p.type}</td>
                <td style={{ paddingTop: '14px' }}>{p.ttl}</td>
                <td style={{ paddingTop: '14px' }}>{p.record}</td>
                <td className='msp-table-button'>
                  <Button bsStyle='warning'
                    disabled={edit.editOn}
                    onClick={() => actions.powerDnsZoneComponent.startRecordEdit(i)}>
                    Edit
                  </Button>
                </td>
                <td className='msp-table-button'>
                  <Button bsStyle='danger'
                    onClick={() => actions.powerDnsZoneComponent.deleteRecord(i)}>
                    Delete
                  </Button>
                </td>
               </tr>
              ))
            }
          </tbody>
        </Table>)
  }
```

Code Snippet 17. Edit Modal component

As seen in Code Snippet 17, React components are a mixture of XML, JavaScript and even CSS. This component is a stateless component, which means that it is a function that takes various parameters, in this case the DNS zone and actions, and returns a React component. It has no state itself, in our application this is handled by a Redux state object. A HTML table is created, and the records are iterated through. If a record is in edit mode the zone information if in input fields, otherwise they are read-only with an edit and delete button. One might realise that the

JSX may have style attributes. These are much like CSS but are actually JavaS-cript objects. For example, the “minWidth” property in the style object of the FormControl components would translate to “min-width” in CSS.

While the previous component is a stateless component, the read-only DNS table was implemented as a class and may be seen in Code Snippet 18.

```
export class PowerDnsDetails extends Component {
  render () {
    const { serviceComponent, actions } = this.props
    const { powerdnsZone } = serviceComponent
    if (!powerdnsZone) {
      return null
    }
    const { records, zone } = powerdnsZone

    // Return component for the PowerDNS zone.
    return (
      <ListGroup>
        <div>
          <ListGroupItem>
            <label>DNS</label>
          </ListGroupItem>
          <Panel
            header={
              <PanelHeader
                onEditStart={() => actions.powerDnsZoneComponent.start(serviceComponent)}
                zone={zone} />
            }
            className='dns-panel'>

            <Table striped responsive>
              <thead>
                <tr>
                  <th>Name</th>
                  <th>Type</th>
                  <th>TTL</th>
                  <th>Data</th>
                </tr>
              </thead>
              <tbody>
                {
                  // Create a row in the table for each record.
                  records.map((p, i) => (
                    <tr key={i}>
                      <td>{p.name}</td>
                      <td>{p.type}</td>
                      <td>{p.ttl}</td>
                      <td>{p.record}</td>
                    </tr>
                  ))
                }
              </tbody>
            </Table>
          </Panel>
          <PowerDnsZoneEditModal resourceView={this.props.resourceView} ser-
viceId={this.props.serviceId} />
        </div>
      </ListGroup>
    )
  }
}
```

Code Snippet 18. PowerDNS read-only table

In Code Snippet 18, this component is not a function, but actually a class, an ECMAScript 6 syntax change. While JavaScript is not an object oriented programming language, one may define prototypes with this kind of syntax. React components must always extend the “Component” class and implement a “render” method which returns a React component. This components also takes a PowerDNS zone and actions as “props”, these are what the XML-like attributes are called, and maps the records as rows in a table.

With the design of MSP, being very modular, a case for PowerDNS only had to be added to a switch statement of resource types for our new component to be drawn as its own resource in the service view.

5.2.2 Resource List

The second kind of component that had to be implemented is the Resource List. This is very similar to the service list that was already implemented, however, this was for single resources. The component may be seen in Code Snippet 19.

```
class ServiceResourceList extends Component {
  componentWillMount () {
    this.props.actions.serviceResourceList.load(false)
  }

  render () {
    const { serviceResources, actions, filter, showSort, provider } = this.props
    const different = []
  const providerFilter = serviceResources.filter((s) => provider !== '' ? s.provider.id ===
provider : true)
  // Filter the service resources according to what is inputted in the search box.
  const filtered = filter === '' ? providerFilter : List(new Fuse(
    providerFilter.toArray(),
   {
    shouldSort: false,
    distance: 1000,
    location: 0,
    threshold: 0.1,
    keys: ['provider.targetId']
   }
   ).search(filter))
   .sort(
    (a, b) => a.provider.targetId.localeCompare(b.provider.targetId)
   )

  // Group resources according to provider.
  serviceResources.forEach((s) => {
   if (!different.includes(s.provider.name)) {
    different.push(s.provider.name)
   }
  })

  return (
   <div>
    <PageHeader className='pageHeader'>Service Resources</PageHeader>
    <Search
     onChange={(ev) => actions.serviceResourceList.search(ev.target.value)}
     value={filter}
     placeholder={'Search with Resource Name or Type'}
    >
     <Col sm={1} md={1} lg={1}>
      <Button onClick={() => { actions.serviceResourceList.toggleSort(!showSort) }}>
       <i className={'fa fa-lg fa-sort'} />
      </Button>
     </Col>

     <Col sm={12} smOffset={0} md={6} mdOffset={3} lg={4} lgOffset={4}>
      <Panel collapsible expanded={showSort} className='sort-panel'>
       <span style={{ fontSize: '125%' }}>Filter by provider:</span>
       <FormGroup>
        <FormControl
         componentClass='select'
         onChange={(e) => actions.serviceResourceList.setFilter(e.target.value)}
         value={provider}
        >
         <option value=''>All</option>
         {different.map((d) => (
          <option value={d.toLowerCase()}>{d}</option>
         ))}
        </FormControl>
       </FormGroup>
      </Panel>
     </Col>
    </Search>
    <Col sm={10} smOffset={1} md={8} mdOffset={2} lg={6} lgOffset={3}>
     {different.map((d) => (
      <ListGroup style={{ marginBottom: '15px' }}>
```

```
              { /* Draw each filtered resource as its own list item.  */ }
              {filtered.filter((s) => (s.provider.name === d))
                .map((r) => (
                  <ListGroupItemLink
                    key={r.provider.targetId}
                    to={'/resources/' + r.provider.targetId}
                    onClick={() => this.props.actions.serviceResource.load(r)}
                  >
                    <div className='service-header' style={{ padding: '10px' }}>
                      <span className='service-header-container'>
                        <label className='service-header-label' style={{ cursor: 'inherit' }}>
                          {r.provider.targetId.length > 37 && r.provider.id === 'openshift'
                            ? r.provider.targetId.substring(0, r.provider.targetId.length - 37)
                            : r.provider.targetId}
                        </label>
                      </span>
                      <span style={{ float: 'right' }}>
                        {r.provider.name}
                      </span>
                    </div>
                  </ListGroupItemLink>
              ))}
            </ListGroup>
          ))}
        </Col>
      </div>
    )
  }
}
```

Code Snippet 19. Resource List

In the previous snippet, a new “class method” is introduced. The “componentWillMount” is another React method which is called when the component is about to be added to the DOM. In our case we use it to query the backend for a list of resources to display. This function is fired before the render method.

Once the list has been loaded, it is filtered based on the text in the search box. After filtering, it is displayed as a list in alphabetical order. Each list item is clickable and when clicked, the target id of the resource is added to the URL, the resource data is loaded from the backend and the resource view is shown.

5.2.3 Resource View

For the resource view, we were able to reuse the code from the service view. Some modifications had to be made so that the component would work in both under a service and as an independent component. A boolean had to be added to differentiate between the two modes and depending on this the API URL end-points would be used for a single resource or a service. This boolean was passed in as a React “prop”. The modified client method for executing an action may be seen in Code Snippet 20.

```
export function sendAction (
  serviceId: number,
  action: any,
  resourceView: boolean
): Promise<any> {
  // Send request to backend.
  return request({
    method: 'POST',
    url: path('api/org', window.localStorage.getItem('context'), 'services', resourceView ? 'ac-
tion' : serviceId),
    body: JSON.stringify(action)
  })
  // Convert the resulting JSON into a javascript object.
  .then((raw) => JSON.parse(raw))
  /* Convert object in to a supported component. Depending on if it is a service or an indi-
vidual service resource */
  .then(s => {
    if (resourceView) {
      return ServiceUtils.serviceComponentFromPlain(s)
    }

    return ServiceUtils.fromPlain(s)
  })
  .catch((error) => {
    console.error(error)
    return null
  })
}
```

Code Snippet 20. Modified client function for executing actions

In Code Snippet 20, the function now takes a Boolean named resourceView and depending on the Boolean, either adds “services” or “action” to the URL. The resulting JavaScript object is also run through a different conversion depending on which endpoint it was return from. For individual resources, the endpoint returns the modified resource while with the service view, the whole modified service is returned.

This is a great example of the reusability of React components. For example, if a button were to be made, it could be implemented as a React component with all the styling and functionality implemented in one place. The component could then take a “click” prop which is a function that would be fired when a user clicks the button. This component could then be used anywhere a button is used with the “<Button>” tag.

6 TESTING

Testing is an important phase during development, which is often overlooked. If a program does not work properly, the project will not be a success. Testing is a quantifiable measure of success. One can define test cases and be able to track progress by how many tests pass. In this project, three different kinds of testing were performed, unit, E2E and user testing.

6.1 Unit Testing

Unit testing is a test where a single method is tested in isolation. Writing unit testable code is a good way to ensure that a single method does not do too much. A unit test executes a method with certain parameters and expects it to return a certain object or value. If it does not return the correct value, the test will error. Unit testing is a valuable tool when refactoring existing code as it works as documentation on what value a method expects and what value is expected to return. In this project, unit tests were configured to run during each build, providing the developer with results between each change. If a test would fail, the whole build fails too. This ensures that all errors will be fixed before the program will compile.

Unit testing was used in this project for testing that the compare method in the PowerDNS. While only one method was tested, multiple tests were written with different cases as input to see that different cases are handled right. An example of a unit test in Java may be observed in Code Snippet 21.

```
@Test
 void withEmptyRecordset_andEmptyUserData_noChanges() {
     PowerDnsClient tester = new PowerDnsClient(bundle);

     List<PowerDnsRecordSet> records = new ArrayList<>();

     List<UserData> userData = new ArrayList<>();

     // Compare empty lists and assume the resulting list is empty.
```

```
        assertEquals(0, tester.compare(zone, records, userData).size());
    }
```

Code Snippet 21. Testing an empty record set

Code Snippet 21 is a simple test in which we feed two empty recordsets to the compare method. The "assertEquals" method is used to declare that we expect the resulting recordset to have a size of 0, therefore also being empty. A more complex unit test may be observed in Code Snippet 22.

```
    @Test
    void withEmptyRecordSet_andOneUserData_markForReplacement() {
        PowerDnsClient tester = new PowerDnsClient(bundle);
        List<PowerDnsRecordSet> records = new ArrayList<>();
        List<UserData> userData = new ArrayList<>();

        String type = "A";
        Long ttl = 60L;
        String content = "127.0.0.1";

        userData.add(buildUserData("@", type, ttl, content));

        List<PowerDnsRecordSet> result = tester.compare(zone, records, userData);

        /* Assume the resulting list have 1 record, that all data matches the userdata and that
the changetype equals REPLACE */
        assertAll(
                () -> assertEquals(1, result.size()),
                () -> assertEquals(zone + ".", result.get(0).getName()),
                () -> assertEquals(type, result.get(0).getType()),
                () -> assertEquals(ttl, result.get(0).getTtl()),
                () -> assertEquals(1, result.get(0).getRecords().size()),
                () -> assertEquals(content, result.get(0).getRecords().get(0).getContent()),
                () -> assertEquals("REPLACE", result.get(0).getChangetype())
        );
    }
```

Code Snippet 22. Testing the replacement of a DNS record

In Code Snippet 22, an empty "existing" record set and a single new record is provided to the comparison method. We then assume that the size of the result will be 1, the data equals the data that we provided in the user data and that the type of action is “REPLACE”. Having many more test cases, a full table of test cases are provided in Table 8.

Table 8. Unit test cases and results

Test Case	Result
With an empty record set and empty user data, no changes.	Success
With a single existing record and empty user data, mark record for deletion.	Success
With an empty record set and one record in user data, mark new record for replacement.	Success
With a single existing record and similar record with different content in user data, mark for replacement with new content.	Success
With an empty record set and multiple user data records, mark multiple replacements.	Success
With a single existing record and multiple user data records, mark one record for replacement.	Success
With an empty record set and user data with a CNAME record, mark for replacement and validate CNAME content formatting.	Success
With similar record set and user data return same changeset.	Success

As may be observed from Table 8, all test cases work and we may be confident that this method works as intended and will continue to work in the future as the build would fail if a test case fails.

6.2 E2E Testing

E2E testing is another way of programmatically testing an application. In our case tests were written, using Selenium, which provides programmatical access to a headless browser and simulates a user interacting with our application. Tests were written to test that everything from the frontend, to the backend and PowerDNS work. The E2E tests are split into two files, the actual tests which define the input and the expectation and a robot file which has methods to interact in certain ways with the browser. The E2E tests were written in JavaScript and use Chromium as the browser for interacting with the application.

The tests themselves are very similar to the unit tests written in Java for the PowerDNS comparison method. A test case, input parameters and an expected output are defined. An example of a test may be observed in Code Snippet 23.

```
it('provisions an empty domain service', async () => {
  const dns = new DNSRobot(driver)
  await dns.get()

  // Create a new service with the name Test DNS
  expect(await dns.createService('Test DNS')).to.equal(true)
})
```

Code Snippet 23. DNS service provisioning E2E test

In Code Snippet 23, the test itself is quite simple. All the interaction is defined in the DNSRobot. The it method is used to define a test and it is provided a function that executes the provided code and an expect method which is used to provide the expected result. The tests are ran using asynchronous promises which are another ECMAScript 6 feature of JavaScript. The corresponding robot method may be seen in Code Snippet 24.

```
async createService (name: string): Promise<boolean> {
  try {
    // Click the new service button
    await this.driver.findElement(
      By.id('floating-action-button')
    ).click()

    const nameInput = await this.driver.findElement(
      By.id('service-create-name')
    )

    await this.driver.wait(
      until.elementIsVisible(nameInput)
    )

    // Input the service name.
    await nameInput.sendKeys(name)

    await this.driver.findElement(
      By.id('service-template-option-domain')
    ).click()

    await this.driver.findElement(
      By.id('service-create-submit-button')
    ).click()

    const service = By.xpath('//*[contains(text(), "' + name + '")]')

    await this.driver.wait(
      until.elementLocated(service)
    )

    // Click the newly created service
    await this.driver.findElement(service).click()

    await this.driver.wait(
      until.elementLocated(
        By.id('service-details-reserved-components')
      )
    )

    const reserved = await this.driver.findElement(
      By.id('service-details-reserved-components')
    )

    await this.driver.wait(
      until.elementIsVisible(reserved)
    )

    try {
      // See if the powerdns activation button is enabled.
      return this.driver.findElement(
        By.id('service-activate-button-powerdns')
      ).isEnabled()
    } catch (_) {}
  } catch (e) {
    console.log(e.message)
  }

  return Promise.resolve(false)
}
```

Code Snippet 24. DNS Robot for provisioning a new service

The provisioning method first finds and click a button with the HTML id of “floating-action-button”. The method then waits for a text input with the id “service-create-name” and send the text defined in a constant at the top of the file and is “DNS Test”. The robot will then click the “Domain” option from the service template list. This is a template which has a DNS service resource reservation included. After this, the robot waits for the new service to appear in the service list and clicks it. The robot will then return true if an activate button with the id “service-activate-button-powerdns” is found on the page and is enabled.

If a robot hangs for too long or an exception is thrown the method will return false. The test cases expect these methods to return true, therefore the test will fail if everything is not working as expected.

Another example of a E2E test is testing the actual DNS server for a record. We can also execute shell commands. In this case we use “dig” to query the DNS server for a certain record. This can be seen in Code Snippet 25.

```
it('adds the record in PowerDNS', (done) => {
  // Check that the record is in PowerDNS using dig.

  exec(
    'dig +noall +answer -t ' + type + ' ' + domain + ' -p 8853 @127.0.0.1',
    (err, stdout, stderr) => {
      expect(
        err
          ? false
          : includesAll(stdout, domain, type, ttl.toString(), data)
      ).to.equal(true)
      done()
    }
  )
})
```

Code Snippet 25. Testing PowerDNS for an added record.

In Code Snippet 25, the “exec” method is used to execute dig and asked for a certain DNS record type and domain from the PowerDNS server running on the local computer on the port 8853. “dig” will return information about the record if it is found and the resulting string may be manipulated with JavaScript. We compare the result to check that it includes the information inputted through the web application in a previous test.

E2E tests are a viable method of testing various changes in code to see that everything still works as intended. As the tests are defined in code, they may also be automated to be run after every build. A full list of implemented test cases may be seen in Table 9 with the result for each test.

Table 9. E2E test cases and results

Test Case	Result
Provisioning an empty domain service in MSP.	Success
Provisioning a new DNS zone in an empty domain service in MSP.	Success
Record gets provisioned in PowerDNS.	Success
Adding a record to provisioned DNS zone in MSP.	Success
Record gets added in PowerDNS.	Success
Editing a DNS record in MSP.	Success
Record gets edited in PowerDNS.	Success
Adding a second DNS record in MSP.	Success
Second record is added to PowerDNS.	Success
Deleting both DNS records in MSP.	Success
Both records are deleted in PowerDNS.	Success

As is shown in Table 9, all test cases are successful and test each component of MSP that are involved with integrating PowerDNS with the application. These tests also allow other developers to work on MSP and remain that this area of the application continue to work in the future as these tests will be ran after each change.

6.3 User Testing

While developing the new features in MSP introduced by this project, continuous integration was used, and new features were deployed to the production environment, which is used by customers. They now manage all DNS zones and records through MSP as well as reportedly use the new service resource list and view to easily search and manage individual service resources. Administrators have also migrated to using the PowerDNS integration in MSP to manage internal DNS through the application's user interface.

Having this type of testing occur throughout the development of the new features ensured that the product works as intended. The list of functionalities with the accompanying results tested is presented in Table 10.

Table 10. Manually tested functionality

Test Case	Result
Creation of DNS zone.	Works as intended.
Deletion of DNS zone.	Works as intended.
Viewing DNS zone and records.	Works as intended.
Adding DNS records.	Works as intended.
Editing DNS records.	Works as intended.
Deletion of DNS records.	Works as intended.
Listing service resources.	Works as intended.
Viewing individual service resource.	Works as intended.
Managing individual service resource.	Works as intended.
Linking existing service resource to service.	Works as intended.

As may be observed all the required functionality has been tested and it works. They have been tested both locally and in production and are used by multiple customers as well as administrators. Together with the unit and E2E tests we can be confident that everything works and will keep working in the future as long as the automated tests are run after each change to check for any breaking changes.

7 SUMMARY

The main objective of the project was to develop a working integration between MSP (Modular Service Platform) and PowerDNS allowing administrators and customers to manage their DNS (Domain Name Service) zones and records through a web application. Smaller objectives were to implement functionality to list and manage individual service resources independent from their parent service as well as log service status changes and actions executed.

The project started by defining the requirements that must be met during the project. After defining the requirements, the design of the components to be implemented was started. The design together with the requirements provided easy to follow guidelines that could be referred to during development. During and after development, testing was conducted for measuring the progress and success of the project itself. All objectives were fulfilled, and the project was a success.

8 CONCLUSIONS

While the goals of the project were achieved, and the implementation works well, there are always challenges and improvements that could be made in every project.

Major challenges during the project were writing code and test cases that were easily testable and conducting E2E (End-to-end) tests. The documentation of the API (Application Programming Interface) for interacting with Chromium were lacking, and many methods did not work as expected, however it was quite a learning experience and in the future, I will use E2E tests as they simulate the activity of a real user, as E2E tests are good for defining the required functionality and easy to run with a headless browser. I also learned a lot about how DNS and newer Java 8 features, such as Optionals and Streams work.

A future improvement to the application could be cleaning up the frontend code as it is quite messy, however, the application itself meets all the specified requirements and works as was intended therefore a success. The new functionality is in daily use by customers and administrators to manage their records.

The next functionality that could be added to MSP is the ordering of new domains as well as creating SSL certificates for the customers web applications.

REFERENCES

/1/ Lehto, N. 2018, Modulaarinen palvelualusta. Accessed 9.4.2018. http://urn.fi/URN:NBN:fi:amk-2017112317953

/2/ Jubic Oy, Jubic Oy. Accessed 9.4.2018. https://www.jubic.fi/en/

/3/ Cloudflare, What is DNS?. Accessed 3.4.2018. https://www.cloudflare.com/learning/dns/what-is-dns/

/4/ Oracle, Java Language Specification. Accessed 9.4.2018. https://docs.oracle.com/javase/specs/jls/se8/html/jls-1.html

/5/ Docker, What is Docker. Accessed 9.4.2018. https://www.docker.com/what-docker

/6/ Fielding, et al., RFC 2616. Accessed 9.4.2018. https://www.w3.org/Protocols/rfc2616/rfc2616-sec1.html

/7/ W3, Web Services Architecture. Accessed 10.4.2018. https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#relwwwrest

/8/ JSON, Introducing JSON. Accessed 10.4.2018. https://www.json.org/

/9/ Dropwizard, Dropwizard GitHub page. Accessed 10.4.2018. https://github.com/dropwizard/dropwizard

/10/ MariaDB, About MariaDB. Accessed 10.4.2018. https://mariadb.org/about/

/11/ jOOQ, jOOQ. Accessed 10.4.2018. https://www.jooq.org/

/12/ Liquibase, Liquibase. Accessed 10.4.2018. https://www.liquibase.org/

/13/ Square Inc., OkHttp. Accessed 10.4.2018. https://square.github.io/okhttp/

/14/ Jackson, Jackson GitHub page. Accessed 10.4.2018. https://github.com/FasterXML/jackson

/15/ Mozilla, Mozilla Developer Network. Accessed 10.4.2018. https://developer.mozilla.org/en-US/docs/Web/CSS

/16/ Mozilla, Mozilla Developer Network. Accessed 10.4.2018. https://developer.mozilla.org/bm/docs/Web/JavaScript

/17/ Microsoft, Microsoft Developer Network. Accessed 10.4.2018. https://msdn.microsoft.com/en-us/magazine/dn463786.aspx

/18/ Bootstrap, Bootstrap. Accessed 10.4.2018. https://getbootstrap.com/

/19/ Facebook, React. Accessed 10.4.2018. https://reactjs.org/

/20/ Internet Systems Consortium, BIND Open Source DNS Server. Accessed 4.3.2018. https://www.isc.org/downloads/bind/

/21/ PowerDNS. Accessed 4.3.2018. https://www.powerdns.com/