Georgi Ossipov

Computer Lab Management System Web Application

Online management system developed with Django

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor’s Thesis

29 November 2014

Author(s) Title Number of Pages Date	Georgi Ossipov Computer Lab Management System Web Application 44 29 November 2014
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialization Option	Web Programming
Instructor(s)	Jaana Holvikivi, Principal Lecturer Tommi Linnakangas, Senior software engineer

The goal of the final year project was to design a web application for managing computers used for software testing. These computers, known as hosts, can be either physical machines or virtual ones. The main function of the developed application was to be able to reserve certain hosts for testing software functionality and to keep track of hosts' software licenses and IP addresses configured for those hosts.

The project was carried out at SSH Communications Security Oyj to help quality engineers with efficient software testing and the company's IT department with available hardware management. The main tools used in this project were Django, a web development framework written in Python programming language, for developing the system's backend. Developing the frontend required more end-user feedback for making the application as easy to use as possible. Technologies used for developing the frontend included HTML, CSS, JavaScript, jQuery and AJAX.

The project has proven to be an efficient way for learning new technologies as well as a helpful tool in internal computer lab management. There is space for improving the application; however the developed functionality and user interface seemed to be sufficient for the company's internal use.

Keywords	Python programming, web-programming, Django framework, testing host management, jQuery, AJAX

Contents

1. Introduction

The object of this report is to describe the design and development process of a web based reservation system. The aim of the project was to implement a web application for managing hosts used in software testing. Software testing is an important part of software development. Most products that are available on the market are thoroughly tested in order to provide customers with the best solutions.

The project was developed at SSH Communication Security Oyj, a company that specializes in network security software. There are different products that are developed at the company and all of those products require testing before they can be deployed at production sites, i.e. clients' network environments. Each product is developed in a separate department and each department has a dedicated group of quality engineers for testing the product. However, there is only one testing environment with hundreds of virtual machines (test hosts) that can be used for testing purposes.

Managing hundreds of test hosts is a tedious task for system administrators (sysadmins) and quality engineers. System administrators have to know what operating system is installed on the host, does the operating system have a valid license, what domain does the test host belong to and what are the overall network configurations of the test host. Quality engineers need to be able to use a test host with a certain operating system and secure shell (SSH) product. It is also important to know that a test host is not being used by anybody else at the same time since that may lead to erroneous performance results. In order to aid the system administrators and quality engineers in their everyday tasks a system for managing a network environment has been developed. In this report I will describe the features and implementation methods of such system.

2. Management System Description

The main purpose of a network environment management system is to keep track of computers that are present in a network environment. Software companies have dedicated test environments for software testing purposes. These test environments are a part of companies' internal networks and may contain large numbers of test hosts which, in turn, are hosted on virtual or physical servers. A single physical server may also contain several virtual servers. The computer lab management system (management system) should help system administrators keep track of all the available virtual servers and point physical machines on which they are hosted. The task of monitoring physical machines in the company falls mostly on sysadmins. However at times they might not have time to add new hardware to the management system due to more pressing matters. In spite of that they need to know when and by whom new physical machines are being added. Adding new objects to the database for the management system is allowed for all users and when a new physical or virtual machine is added the database will also include the user, creation date and time information of the added object.

Quality engineers are responsible for making sure that software that is being tested performs as described to the customer. Different types of software may be used on different operating systems (OSs) with different configurations. Such operating systems as AIX, HP-UX, SunOS, z/OS different Linux distributions and Windows are the main targets for software testing. Each of the listed OS versions has a number of releases and different version releases for the same OS may be used by customers for which the software is targeted. In order to find a test host with the specified OS version and release, filtering functionality for this system has been implemented and will be discussed in section 6.2. However knowing the OS version and release might not be sufficient for sysadmins because they also need to know the vendor of the OS. That information is needed to keep track of the OS licenses purchased from different vendors.

Testing the functionality of network security software is not always limited to using a small number of test hosts. Sometimes it requires thousands of hosts for testing software limitations. A concept of IP ranges was introduced to the management system. An IP range is a number of test hosts with sequential IP addresses. IP ranges are mainly used by quality engineers; sysadmins only take care of adding and configuring said ranges to physical or virtual servers.

A single IP range may contain several thousand of test hosts for stress testing. Stress and regression testing may be divided between several quality engineers and only a part of an IP range is needed by a single quality engineer. Reserving a number of hosts from a specific IP range was added to the management system as well as making the reserved range available for other testers i.e. removing the range reservation.

During the product testing phase a quality engineer needs to know specific test host configurations and change them according to test requirements. If the same test host is being used by more than one quality engineer at a time, it may not be possible to be sure that host configurations are not being changed. Moreover testing software with unneeded or unwanted host configurations might lead to erroneous test results. To guarantee that a test host will not be used by anyone else, reservation functionality has been added to the management system.

Different software products developed at a company have dedicated teams of software engineers and quality engineers for a single product. Each team might be using development and test hosts with specific configurations for developing and testing purposes. In order to easily distinguish hosts between different product teams they are added to different domains that are in use at the company. Although it is not possible to configure new domains via the management system, functionality for monitoring available domains was included in the project.

This chapter focused on describing the main purpose of the management system. The described functionality and its implementation will be discussed in more detail in chapters 5 and 6.

3. Development Environment Setup in Fedora Linux

The management system was developed in the Django web application framework (Django) which is written in Python programming language (Python). As well as Django a number of tools for helping the development process are also Python-based. Python is a general-purpose high-level programming language and it comes with large number of standard library packages. When working on several Python projects, these packages might conflict with each other depending on the framework and Python version. In order to minimize the conflict possibility, it is recommended to use such tools as *virtualenv* and *pip*.

Virtual Environment

A virtual environment is an isolated development environment for a single Python project. Creating virtual environments is possible with a tool called virtualenv. Virtualenv is an open source project that helps Python developers to separate project environments from each other. In order to install virtualenv the operating system should have working installations of Python and pip. Pip is a tool for managing Python packages that are not included in the Python standard library.

The following examples show how to install and configure vitualenv using pip:

```
user@fedora$ sudo pip install virtualenv
user@fedora$ sudo pip install virtualenvwrapper
```

virtualenvwrapper is an additional tool that simplifies interactions with virtualenv. In order to use that tool it should be sourced from the working shell environment in Linux:

```
user@fedora$ source /usr/bin/virtalenvwrapper.sh
```

Virtual environments created with virtualenv are stored locally in the operating system and the tool should know where to look for them. Although the virtual environment location has a default value, it is also possible to use a custom location. In order to achieve that a system environment variable WORKON_HOME can be set to point to the desired location. After installing virtualenv and sourcing virtualenvwrapper, the tool can be used for creating virtual environments:

```
user@fedora$ mkvirtualenv project
(project) user@fedora $
```

Creating a new virtual environment will automatically activate it. To avoid having to specify the virtual environments location and sourcing the virtualenvwrapper script manually each time after starting a new terminal session, a command for doing that can be added to the users ~/ .bashrc script. [1]

Version Control System (VCS)

A version control system is a tool that helps keep track of changes in a file over time. During the development phase of a software project, code changes might cause regressions. Regressions are software bugs that prevent software from functioning as intended. Knowing which changes caused a regression makes debugging easier. There are number of VCSs available such as Mercurial and Git. For this project Git was selected for version control.

Git is one of the most popular VCSs nowadays because of its speed, design and ability to manage large projects. Git is not installed by default on some operating systems, so in order to use it, it has to be installed manually. It can either be installed from source files or using the Linux system package management tool [2]:

```
user@fedora$ sudo yum install git-core
```

After installing the Git VCS, a repository can be created in the project's root directory to keep track of changes done to every file in that directory. To initialize a Git repository the following command needs to be executed from the project's root directory. A number of integrated development environments (IDEs) also have the Git functionality built-in. One of such IDEs is PyCharm which is used for the development of this project.

```
user@fedora$ git init
```

This chapter described the setup for a development environment of the management system. The basics of starting a Django project and the management system design will be discussed in the following chapter.

4. Django Project Set Up

Development of any project starts with specifying the requirements of that project. The main requirement of the application described in this paper is to help system administrators and quality engineers with their everyday tasks. For quality engineers that means locating and reserving test hosts for software testing. For sysadmins the application allows to keep track of IP addresses that are configured for different domains, to manage licenses that are valid or have already expired and to monitor physical or virtual servers and test hosts that are available within the company.

The computer lab management system is a web application developed in Django. Django is a web application framework written in the Python programming language. Installing Django is simplified with the pip tool which was described in Chapter 3. With the pip tool the latest version of Django can be installed by executing the next command from the command line terminal:

```
user@fedora$ pip install Django
```

However this project was not developed in the latest Django version. The required version can be specified after the packaged name. The following command will show how to install version 1.6.5 of the framework:

```
user@fedora$ pip install Django==1.6.5
```

In web applications most of the information, if not all, is stored in a database. Django supports a number of database engines, such as Oracle, MySQL, PostgreSQL, and SQLite. The database engine used during the development process is not necessarily the same engine as used in the production environment. Knowing this the developer should keep in mind possible data type inconsistencies between different database engines. Neglecting to do so may result in various software bugs in production that have not been encountered during development. [3]

To make the transition between development and production environments less error prone, the PostgreSQL database was chosen for both environments. PostgreSQL is an open-source database engine that was developed by a group of volunteers and the development was supervised by RedHat and EnterpriseDB. Even though PostgreSQL is an open-source database management system, a number of security extensions are available for purchase from EnterpriseDB. [4]

Django Project Layout

After creating a virtual environment and installing Django a new project can be started. To create the project base the following command needs to be executed:

```
user@fedora$ django-admin.py startproject <project_name>
```

Executing this command will create a directory with files that are required for a Django project to work. The layout of a Django project may vary depending on developers' preferences. The initial project layout is illustrated in Listing 1:

```
<project_root>/
    manage.py
    project/
        __init__.py
        settings.py
        urls.py
        wsgi.p y
```

Listing 1. Initial Django project layout

The items listed above are described below:

- <project_root> - Root directory of the Django project which is also the root of the GIT repository
- manage.py - Command line utility for performing administrative tasks on the given project. [5]
- project/ - Global project directory containing files which are used by apps in the project.
- __init__.py - A Python file that allows the directory to act as a Python package
- settings.py - Global project settings file
- urls.py - File containing global Uniform Resource Locators (URLs) used in the project. May also be used for separating URLs for different apps.
- wsgi.py - Web Server Gateway Interface used by the built-in Django server which is used during development.

The initial Django project can be used for creating static websites. However if there is a need to create more complex web pages an app can be added to the project. The manage.py file incorporates the functionality for starting apps in a Django-based web site:

```
user@fedora$ python manage.py startapp app
```

Executing this command will create another directory in the project root folder with the app's name. Listing 2 describes the Django app structure:

```
app/
    __init__.py
    admin.py
    models.py
    views.py
```

Listing 2. Django app layout

The items listed above are described below:

- app/ - A web app directory that stores app specific files.
- __init__.py - See Table 1.
- admin.py - An app specific administrating Python file which is used for registering Django models in the built-in admin site.
- models.py - A file for storing models specific to the given app.
- views.py - A file for writing views or code for rendering templates.

Even though all the necessary files and directories for the project are created the project still needs to be configured. The global project configurations are read from the global setting file mentioned in Listing 1. The setting.py file is used for setting up, for example, the projects database engine, template directories, static file finders and email servers. Large projects may contain several apps which in turn may use different databases, dozens of models and large numbers of templates. In those situations it is possible to configure different setting files for different apps. However local app settings are out of the scope of this project and a single global settings file is used for this project.

Django Database setup

The Django web application framework supports four different database engines, such as Oracle, MySQL, PostgreSQL and SQLite. The project's database engine is set up by providing the database information as a Python dictionary variable DATABASES. Starting from Django version 1.6, the initial settings.py file has SQLite configured as the project database upon starting the project. However, since the production environment was set up with PostgreSQL the same database engine was configured for the development environment. The setup of the database engine is illustrated in Listing 3:

```
DATABASES = {
    'ENGINE':  "django.db.backends.postgresql_psycopg2",
    'NAME': "labhosts",
    'USER': "admin",
    'PASSWORD': "<password>",
    'HOST': "",
    'PORT': "",
}
```

Listing 3. Django database engine settings

The items needed for database configuration are described below:

- ENGINE key denotes which of the supported database engines is used,
- NAME key tells the name of the database for the project,
- USER key shows who is the owner of that database
- PASSWORD key provides the database owner's password for accessing the database.

The HOST and PORT may be left empty if the database is run on the local machine and is using the default port for database connectivity.

When setting up the database one should remember that Python needs an adapter in order to interact with the database. For interacting with PostgreSQL a Python module psycopg2 is required. Like other Python modules in a virtual environment, psycopg2 can be installed using pip. However to streamline the installation of the module environment variable PATH has to contain the location of pg_config program which comes with the database development package.

Django Built-In Apps

The Django framework comes with a number of useful built-in applications that can be integrated in web applications. One of such apps is the Django administration app. The admin app makes it fairly straightforward to manage the content of the project's database. The built-in web interface allows the developer to see the project models and to add or remove object from the database. Another useful app is the authentication app. The authentication app provides the developer with tools for managing the users of an application and authenticating the users to the application.

For managing users in the reservation system the Django's build in authentication app is used. With the help of this app it is possible, for example, to create and/or delete users, assign one of the three available user roles, or grant or revoke user permissions to perform different types of action on database objects. The three available user roles are:

1. Active – designates whether the user is active. If the user is not active it is treated as deleted. Active status is not convenient for users of the reservation system. At some point users will want to change their password, for example. For that they will need to have permissions to log in to the admin site. For that purpose the staff role is going to be assigned to all users in the system.

2. Staff Status – designates whether the user can log in to the Django built-in admin site. Users with this role are not able to alter any database tables from the admin site. All users of the system will have the role of a staff member in order to provide them with the option to change their password.

3. Superuser Status – designates that the user has permissions to perform all the available actions on Django models used in the web application without explicitly assigning those. (Description of Django models will follow later in section 5.1). All users who should be able to add new users, delete users that no longer need an account, assign licenses to hosts and keep track of all the changes made in the database will have a superuser role. Such users include project owners, team leaders and members of the IT support department.

After the database has been set up and all the required models have been created the database has to be synchronized with the project. To synchronize the database from the command line, Django's manage.py utility is used:

```
user@fedora$ python manage.py syncdb
```

The command will create the necessary tables for the created Django models and apply field properties defined in those models such as length of the varchar fields and default field values. Because the administration app is installed by default Users table will also be created, and an option for creating a superuser for the admin app will be available.

Templates

In order to have a functioning web site, having information in the database is not enough. The data needs to be presented to the user. Being a web application the information is displayed in HTML pages (templates). Before creating any templates several variables need to be configured in the global project's settings file. First of all the framework needs to know where to look for templates. By default Django tries to find templates in each application's templates subdirectory. However if those templates are "extended" to a single base.html file the location of the base.html template needs to be configured in the global setting file with TEMPLATE_DIRS variable. The default settings file has a BASE_DIR variable defined that points at the location of the project's root directory in the operating system. A directory and operating system location processing tool is included in Python operating system (os) module [5]. Before specifying the location of the base template a directory *templates* was created in the project's root directory. Part of the settings.py file that handles the TEMPLATE_DIRS variable is illustrated in Listing 4:

```
import os
BASE_DIR = os.path.dirname(os.path.dirname(__file__))
# points to the project root directory
TEMPLATE_DIRS = os.path.join(BASE_DIR, 'templates')
```

Listing 4. Django based web application template directory configuration

Static files

Django templates are simple text files that are presented to the user in a web browser. In order to make the templates more appealing from the user’s perspective cascading style sheets (CSS) and JavaScript (JS) are applied to the templates. Static files are usually stored in a separate directory depending on the developer’s preference. If a project consists of a single application, the static files directory can be created within that application’s directory. As with templates Django will look for static files in the application’s *static* subdirectory.

However in bigger projects with several apps, creating and maintain static files in several locations is tedious and needs more attention. In that case a static file directory can be placed as a subdirectory under the project’s root. This also requires changes in the global settings file. In order to include such files as cascading style sheets (CSS), JavaScript (JS) libraries and images in templates a variable STATICFILES_DIRS can be added to the settings.py file. STATICFILES_DIRS is a Python tuple variable which contains strings that point to possible locations of the static files’ directories. An example of STATICFILES_DIRS variable is illustrated in Listing 5:

```
STATICFILES_DIRS = (
    os.path.join(BASE_DIR, ‘static’)
)
```

Listing 5. Django based web application static files directory configuration.

The STATICFILES_DIRS variable points to the static files directory that is created under the project’s root. Because the management system is a single application Django project, the static files directory was created in the application’s main directory. The final project structure discarding additional Python libraries for back-end functionality is displayed in Listing 6:

```
<project_root>/
    app/
        templates/
        static/
        __init__.py
        admin.py
        models.py
        views.py
        manage.py
    project/
        __init__.py
        settings.py
        urls.py
        wsgi.py
```

Listing 6. Management System Project structure

This chapter was focused on describing the initial setup of a Django project. Additional libraries were not included because they will be discussed in chapter 5 and changes in the project structure will also be mentioned, if applicable.

5. Back-end Development

This chapter will focus on describing the development of the required components for the management system. These components include object models that will be stored in the database, functionality for rendering web pages with information stored in the database and forms for creating and updating objects. The project was started as an update for the already existing host management system. After having completed the basic application functionality, several employees from different departments of the company were consulted and additional features for the application were requested.

5.1. Django Models

A model is a Python object that is used for data persistence. Django has a number of Python classes that are inherited in self-created Python objects. All models defined in the models.py file inherit the predefined Django Model class [6]. Because some models are quite large, simplified examples of these models will be presented in this chapter.

First of all the management system has to contain information about the available hosts for software testing. In order to decide whether the host is suitable for the quality engineer’s testing purposes, the minimal required information about the host includes the host name, IP address, operating system, SSH product that is installed on the host, whether the host can be used and other miscellaneous information that could be of use. A simplified version of the host model is illustrated in Listing 7:

```
import django.db.models
from django.contrib.auth.models import User


    class Host(models.Model):

        th_name = models.CharField(max_length=50)
        th_ipv4 = models.IPAddressField()
        th_os_vendor = models.ForeignKey('PlatformVendor')
        th_os_version = models.ForeignKey('PlarformVersion')
        th_os_release = models.ForeignKey('PlatformRelease')
        th_os_license = models.ForeignKey('License')
        th_has_license = models.BooleanField()
        th_notes = models.TextField()
        th_ssh_server = models.CharField(max_length=50)
        th_ssh_server_state = models.NullBooleanField()
        th_res_status = models.BooleanField()
        th_reserved_by = models.ForeignKey(User)
        th_reserved_until = models.DateField()
```

Listing 7. Simplified Host model

As mentioned at the beginning of this chapter all models inherit the Django Models class. The Model class is defined in the django.db.models module as well as other classes for representing the database field types. Most of the model field types are self-explanatory according to the field name and the required field parameters are included in the example. The ForeignKey field, however, requires some detailed explanation. The foreign key field stores the value primary key value of an object from another model. Additional models that were created to support the Host model include:

- PlarformVendor – a model for storing information about the platform vendor
- PlarformVersion – a model for storing information about the platform version of the corresponding platform vendor
- PlatformRelease – a model for storing information about the platform release of the corresponding platform version

The *th_reserved_by* field points to the framework's built-in User model. As seen from the import statement, the User model comes from the *django* package which is the main package of the Django framework.

Being a subclass of the Django Model class the Host model inherits all of the Model methods. When saving a model to the database the models *save* method is called. This method can be replaced if needed. There are three fields that store the reservation information: *th_res_status*, *th_reserved_by* and *th_reserved_until*. As the field names suggest they represent the user for whom the host is reserved and the date until which it is needed by that user. The *th_res_status* Boolean field, however, was added for filtering purposes. Representation of host information and filtering functionality in the front end will be discussed in chapter 6. The *th_res_status* flag is needed to know if the host is reserved or not. For this purpose the save method of the Host model has been overwritten and is illustrated in Listing 8:

```
def save(self, *args, **kwargs):
    if self.th_reserved_by is None:
        self.th_res_status = False
    else:
        self.th_res_status = True
    super(Host, self).save(*args, **kwargs)
```

Listing 8. Host model save method

As the displayed method suggests, if the host is not reserved the *th_res_status* flag is set to *False* otherwise it is *True*. A similar Boolean field’s manipulation has been implemented in the License model. The License model was added to the management system for managing different types of licenses that are purchased by the company. A simplified version of this model is illustrated in Listing 9:

```
class License(models.Model):
    license_title = models.CharField(max_length=30)
    license_name = models.CharField(max_length=30)
    license_notes = models.TextField)
    license_valid = models.BooleanField()
    license_valid_until = models.DateField()
```

Listing 9. Simplified License model

Managing license can be a tedious task depending on the amount of license that is currently used. Although when adding a new license to the system the *license_valid_until* is a required date field, checking the validity of each license by going through the whole license list is a burdensome task. In order to see a list of all invalid licenses a Boolean flag *license_valid* was added to the model. The backend was programmed to check all date-related fields once every 24 hours. However we also need to know if the added license is valid. The overwritten *save* method of the License model is illustrated in Listing 10:

```
from datetime import date

    def save(self, *args, **kwargs):
        if self.license_valid_until < date.today():
            self.license_valid = False
        else:
            self.license_valid = True
        super(License, self).save(*args, **kwargs)
```

Listing 10. License model save method

The Python datetime module allows the developer to process dates. When a new license is being added, before saving it into the database the model’s save method checks the license validity and sets the *license_valid* flag to the respective state.

Testing software functionality at times requires a large number of test hosts, even thousands. For the purpose of managing large test environments such models as Host Group (HG) and Host group Reservation (HGRes) models were added to the management system. The HG model contains information about a range of IP addresses of test hosts and the HGRes model holds information regarding the IP range reserved by a quality engineer from a certain host group. In order not to overload the database with thousands of IP addresses both models contain information about the start and end IP addresses and the processing of these ranges is done with a Python module named *netaddr*. The *netaddr* module is used for form validation and will be discussed in Django Forms subsection of this chapter. A simplified version of the HG model is illustrated in Listing 11:

```
class HG(models.Model):
    hg_name = models.CharField(max_length=30)
    hg_host = models.ForeignKey('VMContainer')
    hg_os_vendor = models.ForeignKey('PlatformVendor')
    hg_os_version = models.ForeignKey('PlatformVersion')
    hg_notes = models.TextField()
    start_ipv4 = models.IPAddressField()
    end_ipv4 = models.IPAddressField()
```

Listing 11. Simplified HG model

When thinking of an IP range one can imagine that all that is needed is the start and end IP addresses. However there might be several IP ranges used in a company and the easiest way to refer to hem would be a name for an IP range. Every IP in a host group points to a virtual host that is hosted on a physical server. Detailed information about the fields used in this model is described below:

- hg_name - Name of a host group agreed upon by quality engineers
- hg_host - Physical server's host name on which the virtual hosts are configured
- hg_os_vendor - Foreign key fields pointing to the PlatformVendor model
- hg_os_version - Foreign key pointing to the PlatformVersion model
- hg_notes - Miscellaneous information regarding the host group
- start_ipv4 - The first IP address in the host group
- end_ipv4 - The last IP address in the host group

The host group reservation model (HGRes) is similar to the host group model, but does not include the OS information, because it only refers to the host group. The HGRes model simplified version is illustrated in Listing 12:

```
class HGRes(models.Model):
    hg = models.ForeignKey('HG')
    reserved_by = models.ForeignKey(User)
    start_ipv4 = models.IPAddressField()
    end_ipv4 = models.IPAddressField()
    reservation_size = models.IntegerField()
```

Listing 12. Simplified HGRes model

This model required additional validation. First of all the reserved IP range cannot have IP addresses that do not belong to the group in which the reservation is made. Second, the reserved range cannot overlap with any existing reservation because, as mentioned in chapter 2, this may lead to several users using same hosts and giving erroneous test results.

As mentioned in chapter 4 subsection “Django Built-in Apps”, Django framework comes with a built-in administration app. This app can be enabled or disabled in the *settyngs.py* file. When staring a new Django project the admin site is enabled by default. To disable that functionality in the settings file the ‘django.contrib.admin’ string should be removed from the INSTALLED_APPS tuple or commented out. Enabling the admin site, however, will not automatically give access to the models defined in *models.py*. To access those models they first need to be registered to the admin site. Registering models is done by adding them to the *admin.py* file. An example of this file's content is illustrated in Listing 13:

```
from django.contrib import admin
from app.models import Host

admin.site.register(Host)
```

Listing 13. Example of registering a model to be accessible from the admin site

The models describes in this chapter are the ones that require additional validation and back-end functionality. For example the host’s SSH server checking, license validity checking, host’s reservation period checking and host group and host group reservation overlapping checking. All these features are implemented in Django forms and the management system’s backend and will be discussed in more detail in later sections 5.3 and 5.5.

5.2. Django Views

Django Views contain information that will be displayed in a web browser. There are two types of views that can be used in Django: function-based (FBC) views and class-based views (CBV). FBVs are simple Python functions that take an HTTPRequest object as a variable and return an HTTPResponse object. The HTTPResponse is basically a rendered HTML code with the provided variables and content type. Although using FBVs might seem cumbersome they are still useful in a number of cases. The CBVs are Python object that have the main functionality for rendering templates built-in and in the simplest cases require only a template name as a variable.
Both CBVs and FBVs are used in this project and examples will be given in this chapter.

Django has a number of predefined view classes that can be inherited in user defined views. Generic views used in this project include ListView, DetailView, CreateView, UpdateView and TemplateView. Descriptions of these views are listed below:

- ListView – A view that is used for displaying a number of database entries in a template
- DetailView – A view that is used for displaying information about a single object
- CreateView – A view that is used for adding new objects to the database
- UpdateView – A view that is used for updating existing objects in the database
- TemplateView – A simple view that renders a template with provided variables

These views require different attributes for functioning as intended. For example the framework needs to know which template to use when rendering the view. The *template_name* attribute is a Python string that holds the name of the '.html' file. After providing a template file name the framework will search all the configured TEMPLATE_DIRS directories for a file with that name and if it is not able to find that file a *TemplateDoesNotExist* exception will be raised.

In order to make a Django-based web site dynamic there is a number of ways for injecting user specific data into the template. The main functions for doing so are *render()* and *render_to_response()*[6]. When rendering a template to be displayed in a web browser each of the generic class-based views mentioned above calls the *render_to_response()* method. This method accepts a Python dictionary as context data and renders the template with that dictionary's key, value pairs. A simple example of this method is illustrated in Listing 14:

```
from django.shortcuts import render_to_response

def example(request):
    heading = 'Page heading'
    return render_to_response(
           'template.html',
            {'heading': heading},
            content_type='text/html')
```

Listing 14. FBV example of the render_to_response() method

The above function will render the *template.html* template replacing any occurrence of the *heading* variable with the '*Page heading*' string. When a class based view calls this method it gets the Python dictionary with specified variables from another method called *get_context_data()*. To provide additional data to be rendered in a template can be overwritten in the view code. The same example for rendering a template with the *heading* variable is illustrated in Listing 15:

```
from django.views.generic import TemplateView

class Example(TemplateView):
    template_name = 'template.html'

    def get_context_data(self, **kwargs):
        context = super(Example, self).\
                get_context_data(**kwargs)
        context['heading'] = 'Page heading'
        return context
```

Listing 15. CBV example

The outcome of both examples is the same: adding the *heading* variable to a template. Although the CBV example seems to have more code it simplifies the overall view handling. More examples of the class-based view will be given later in this chapter.

List views, as described in the beginning of section 5.2, are used for passing a list of model instances to a template. Overall there are five list views in the management system web application. In the management system list views were added for displaying domains that are used in the company, servers which are used for installing test hosts, test hosts that can be used for testing purposes, IP ranges for scalability testing and licenses. Creating a list view for a model is fairly easy with class based views. Besides the template name the view needs to know which model's objects it is generating a list for. An example of the list view for test host objects is illustrated in Listing 16:

```
from django.views.generic import ListView
from app.models import Host

class HostListView(ListView):
    template_name = 'host_list.html'
    model = Host
```

Listing 16. List view example

To get a list of objects to be sent to the template the view executes the *get_queryset* method. By default this method returns a list of all database entries in the table containing host objects. Filtering these entries by values specified in the GET request of a URL can be achieved by rewriting the *get_queryset* method. The *get_queryset* method applied to the HostListView is illustrated in Listing 17:

```
    def get_queryset(self, **kwargs):

        filter_dict = {}
        filter_request = dict(self.request.GET)

        if filter_request:

            [filter_dict.update({key: value[0]})
             for key, value in filter_request.iteritems()]

            queryset = self.model.filter(**filter_dict)

        else:

            queryset = self.model.objects.all()

        return queryset
```

Listing 17. *get_queryset* method of the HostListView

As seen from the example above there is nothing that would identify the view class for which this filtering method is applicable. Because each ListVew requires a model attribute, the model attribute can be referenced with *self.model* which holds the name of the actual database model. Instead of rewriting the same code in each ListView a '*mixin*' for filtering all list views was created. A '*mixin*' is a Python class with certain functionality that can be inherited by other classes where that functionality is needed [7]. The final version of the HostListView along with the filtering mixin is illustrated in Listing 18. The example, however, does not include the additional variables that are sent to the filter template.

```
from django.views.generic import ListView
from app.models import Host

class ListFilterMixin(object):

    def get_queryset(self, **kwargs):

    filter_dict = {}
    filter_request = dict(self.request.GET)

    if filter_request:

        [filter_dict.update({key: value[0]})
         for key, value in filter_request.iteritems()]

        queryset = self.model.filter(**filter_dict)

    else:

        queryset = self.model.objects.all()

    return queryset


class HostListView(ListFilterMixin, ListView):
    template_name = 'host_list.html'
    model = Host
```

Listing 18. HostListView with filtering mixin

Detail views are used for displaying information of a single object in the database table. From the end user perspective having all the available information on a single page is not appealing. For the purpose of allowing the user to select an item from a list and check only that item’s details, each object on a web page contains a link to the objects detail page. In order to know which object to fetch from the database the object’s primary key needs to be provided to the view. This key is given in the URL pointing to the detail page. More detailed information about URLs will be given in section 5.4.

Create and update views are used for adding new entries to the database and modifying already existent ones. Both views render a template with an HTML form. Required attributes for these views include *form_class* and *success_url*. The *form_class* variable holds the form class name that will be rendered on a template and the *success_url* is a URL to which the user will be redirected after successful form submission.

This chapter focused on the coding part of Django views. The framework comes with a number of useful predefined Python classes for implementing template rendering in the back-end. More details of the front end part of the application will be discussed in the chapter 6.

5.3. Django Forms

Although Django framework comes with a built-in admin site, not all users are able to access it. Every user of the application should be able to add, for example, test hosts to the management system. New objects can be added to the database or existing ones updated with Django forms. Django forms, as anything else in the framework, are Python classes that incorporate functionality not only to display an HTML form elements in a template but also validate the field data that is entered by the user.

Django has two types of form classes: the simple Form class, in which the all the fields are specified manually and the ModelForm class, which constructs a form according to a model's fields. In the management system application objects that will be accessible to the user in the web browser are: domains, physical and virtual servers, test hosts, host groups and licenses. Because all of these objects already have corresponding models, forms for them have been added as ModelForms.

Creating a form based on a model the ModelForm class, much like a list view needs to know the model, according to which to construct a form. Metadata of the model is provided in a nested class of the ModelForm class. An example of the ModelForm class created for the Host model is illustrated in Listing 19:

```
from django import forms
from app.models import Host

class HostForm(forms.ModelForm):

    class Meta:
        model = Host
        fields = ['th_name',
                  'th_ipv4',
                  'th_server',
                  'th_os_vendor',
                  'th_os_version',
                  'th_os_release',
                  'th_os_license',
                  'th_notes',
                  ]
```

Listing 19. ModelForm for the Host model.

The *fields* attribute in the *Meta* class of the form is a list of model fields that will be displayed in the form template. Because not all model fields can be edited by the user when adding or updating a host, such as the host's SSH server state or reservation details, those fields are not included in the *fields* list.

The CreateView and UpdateView views require a form class to render a form on a template. The same form can be used for both adding new objects to the database or editing existing objects. To avoid typing the same code twice the example of the above form can be used for either purpose. However when updating exiting host data there are fields that should not be edited, such as the IP address that is configured on the host. As any Python class the ModelForm calls the *__init__* method for initial data assignment. This method can also be used for manipulating the form field attributes in HTML code. To make a HTML <*input*> tag not editable it can be given a *readonly* attribute. To separate the forms for creating and updating hosts two more ModelForms were created. These forms are illustrated in Listing 20:

```
class HostAddForm(HostForm):
    pass

class HostUpdateForm(HostForm):

    def __init__(self, *args, **kwargs):
        super(HostUpdateForm, self).\
              __init__(*args, **kwargs)

        self.fields['th_ipv4'].widget.\
                               attrs['readonly'] = True
```

Listing 20. Forms for creating and updating Host objects

Both of these forms inherit the HotsForm class which holds the model and fields information. The only difference is the initial data in the form. Like the DetailView, the UpdateView that renders the HostUpdateForm requires the objects primary key for getting initial form data from the database.

Although Django form classed have default field validators, they only validate the field values according to specified field attributed. In some cases additional validation is required. For example the IP ranges. A starting value of a range cannot be greater than the end value also a new range of IP addresses cannot overlap with an existing one. As well as the IP range reservations. The IP range reservation needs to be within the range of a host group and cannot overlap with any existing reservation. Because the same validators can be used by different forms all additional validators were implemented with Python *mixins*. Managing IP addresses is simplified with the Python *netaddr* module. This module allows the developer to manipulate IP addresses, for example such actions as comparison, mathematical operations, working with network masks and IP address ranges are simplified with the use of this module.

According to the mentioned validation requirements several *mixins* were added to support forms related to host group functionality. An example of one of those mixins is illustrated in Listing 21:

```
class ValidIPRangeMixin(object):

    """
    Mixin for checking if the host group start IP is not
    greater than end IP
    """

    def clean(self):
        cleaned_data = super(ValidIPRangeMixin,
                             self).clean()

        start_ipv4 = cleaned_data.get('start_ipv4')
        end_ipv4 = cleaned_data.get('end_ipv4')

        validate_range(start_ipv4=start_ipv4,
                       end_ipv4=end_ipv4)

    return cleaned_data
```

Listing 21. Form validation *mixin* example

In Django forms the *clean* method is called before the actual validation occurs. This method returns a Python dictionary with keys being the form fields names and values being the data inside those fields. After that the dictionary is validated with default validators. In this example the start and end IP addresses are checked before sending the form data to validators. Checking is the start IP is indeed lower than the end IP a helper function for validating a range was added to the core functionality of the project. The *validate_range* function is illustrated in Listing 22:

```
def validate_range(start_ipv4, end_ipv4):

    if start_ipv4 and end_ipv4:

        try:
            start_ipv4 = netaddr.IPAddress(start_ipv4)
            end_ipv4 = netaddr.IPAddress(end_ipv4)
        except netaddr.AddrFormatError as error:
            raise error

    if start_ipv4 > end_ipv4:
        raise ValidationError((suctom_error_message))
```

Listing 22. Function for validating range of IP addresses

This function accepts two values from the form: the start IP and the end IP fields’ data. After that it transforms that data into *netaddr* module’s *IPAddress* objects and compares them. If the start IP address is greater than the end IP address a validation error is be raised and the object is not saved to the database.

This chapter explained the main purposes of the Django's ModelForm class and its usage. How the forms are sent to the server for processing and validation errors are displayed on a web page will be discussed in chapter 6.

5.4. Django URLs

Django framework comes with a built-in development server. It can be used during the development phase of a project for testing purposes. The server can be started with the help of *manage.py* command line utility.

```
user@fedora$ python manage.py runserver
```

Running this command will start the development server on the computers loopback address port 8000. Entering that address in a web browser will display a built-in Django page saying that the server is running. To let the framework know which URL is to be used to access what view a list of available URL addresses needs to be specified in the *urls.py* file.

The available URLs for the project is generated by the Django's *patterns* function an returns a list of addresses that can be used to access the server. An example of the *urls.py* file is illustrated in Listing 23:

```
from django.conf.urlsimport patterns, url


urlpatterns = patterns(
          url(r'^host/list/$', HostListView.as_view(),
                               name='host-list'))
```

Listing 23. *urls.py* file example

By default Django framework checks the *uelpatterns* list to find a match to the address entered in a web browser. If a match is found the view specified after the URL string is rendered and displayed as a web page. The *name* is a string variable that can be referenced in HTML hyperlinks to access the page with that view.

5.5. Back-end Functionality

The backend in this project was designed to check the SSH server status on all hosts that were added to the management system as well as keep track of effective test host reservations and license validity. To control the back-end a process control system *supervisor* was used. With the help of *supervisor* it is possible to manage processes on Unix like systems [8].

Checking the status of SSH server on hosts in the system a Python *socket* module was used. This module makes it possible to connect to a remote SSH server and get information about that server. The function for checking connectivity to a host is illustrated in Listing 24:

```
import socket

def get_host_server(host_ip, port=None):

    try:

        port = port
        sock = socket.socket(socket.AF_INET,
                             socket.SOCK_STREAM)
        sock.settimeout(5)
        sock.connect((host_ip, port))

        response = sock.recv(1024)

        sock.close()

    except socket.error:
        response = None

    return response
```

Listing 24. Function for checking a remote host’s SSH server

First a socket object is created with the specified IP address and port number. The socket tries to establish a connection to a host with that IP address. If the connection is successful the socket received a string response with the running SSH servers name. If for some reason the socket cannot connect to the remote host for five seconds it eill stop trying to connect and return nothing to the caller function signifying that no SSH server is running on the host with the specified IP address.

Checking reservation and license validity is somewhat different. Test host reservations are made until a certain date as well as licenses can be valid until a certain date. So to change the status of a test host to ‘Available’ the back-end needs to check all reservations once a day and change their status. The reservation checking function is illustrated in Listing 25:

```
def check_reservations():

    expired_res_ids = Host.objects.filter(
                      th_reserved_until__lt=date.today()).\
                      values_list('id', flat=True)

    for pk in expired_res_ids:

        host = Host.objects.get(pk=pk)
        host.th_reserved_until = None
        host.th_reserved_by = None
        host.save()
```

Listing 25. Host reservation checking function

First the back-end collects primary key values of all hosts that were reserved until the previous day. Then looping through the list it changed the reservation details of that host to *None*. Almost the same function was implemented for checking license validity. The only difference is that for licenses the *license_valid* flag is set to *False*.

The purpose of this chapter was to describe the back-end functionality of the management system.

6. Front end Development

Developing the front-end of an application requires thinking not only from the developer’s point of view but also from the end user perspective. A front end needs to be straightforward and understandable. Simple web pages containing text and a number of links are not appealing. In this chapter such front-end frameworks and tools as *Bootstrap*, *jQueryUI* and *toastr* will be discussed.

6.1. Project Templates

Web application templates are simple HTML files that are displayed in a web browser. Django views that display information about different model objects require their own templates. In this project each ListView has a dedicated template. However not every object in a list needs a separate template. Because each object belongs to a certain model, the same template can be used to display detailed information of different objects that belong to the same model.

To display any form for adding or updating an object a single template will suffice. As mentioned in section 5.3 CreateView and UpdateView are views that require a form class to be rendered in a template. Form data for all form-based views is sent to a template with the same *form* variable. So a single template can be used for any types of forms used in the project.

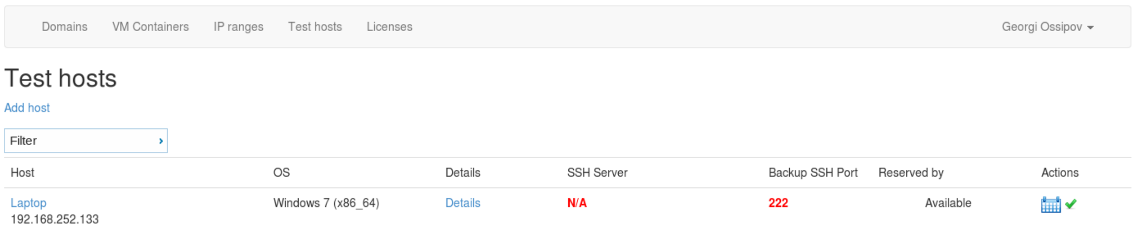
6.2. Host List Templates

ListViews render a template with a number of database entries. As mentioned in section 5.2 the ListView class gets a list of object from the database table of a model and sends these objects to the template. The list can then be iterated through for creating a separate HTML tag for each object. A simplified example of iterating though a list of Host objects is illustrated in Listing 26:

```
<table>
  {% for host in host_list %}
    <tr id={{ host.pk }}>
        <td>{{ host.th_name }}</td>
        <td>{{ host.th_ssh_server }}</td>
        <td><a href="{% url 'app:host-detail' host.pk %}">
            Details
            </a>
        </td>
        <td>{{ host.th_reserved_by }}</td>
    </tr>
  {% endfor %}
</table>
```

Listing 26. Host list iteration example

The above example does not include all table columns that are applied to the table or any additional HTML elements that are used. They will be discussed if applicable. The end result of the host table is displayed in Figure 1:



Domains VM Containers IP ranges Test hosts Licenses Georgi Ossipov

Test hosts

Add host

Filter

Host	OS	Details	SSH Server	Backup SSH Port	Reserved by	Actions
Laptop 192.168.252.133	Windows 7 (x86_64)	Details	N/A	222	Available	

Figure 1. Rendered test host table

At the top of the page is a navigation bar for listing the corresponding objects. The right corner of the navigation bar is the logged in user full name which acts as a drop down menu. The user menu allows the user to change their password or check the user's current reservations. The use drop down menu is displayed in Figure 2:

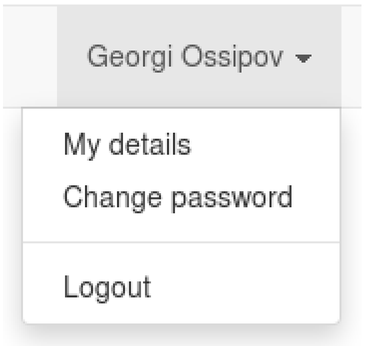
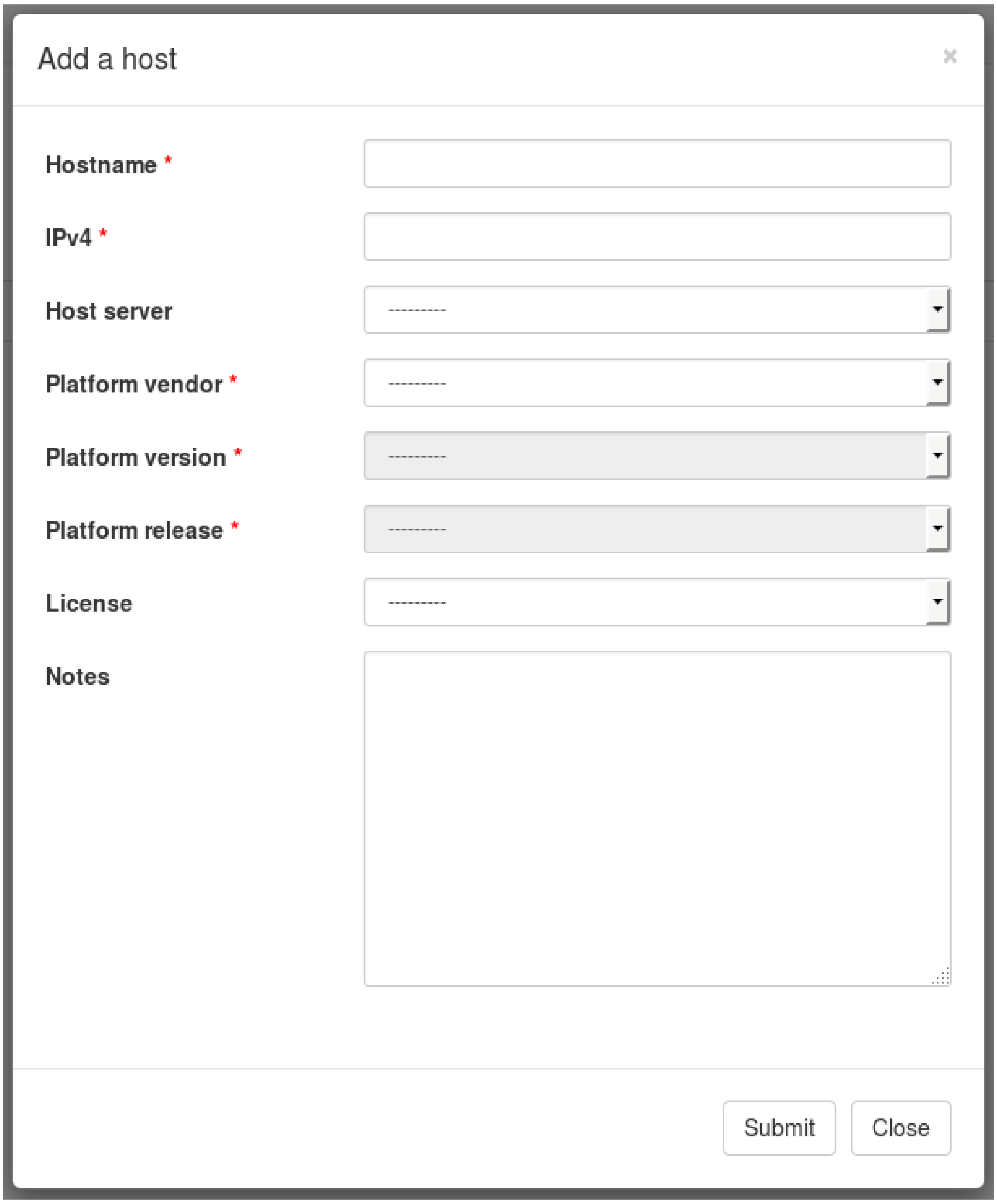


Figure 2. User drop down menu

Above the host table are the page heading, a link for displaying a form for adding new hosts and the filter list. Clicking the “Add host” a Bootstrap component *modal* with the host form is invoked. Loading *modal* data is already built in to the framework functionality. The requirements for modals to be loaded correctly are the HTML hyperlink tag needs to have a *data-toggle* attribute which is equal to *modal* and it should point to URL with the desired data. A simplified version of the host form is displayed in Figure 3:



Add a host

Hostname *	
IPv4 *	
Host server	---------
Platform vendor *	---------
Platform version *	---------
Platform release *	---------
License	---------
Notes	

Figure 3. Form for adding hosts in Bootstrap modal

Form processing and invalid input examples will be provided later in chapter 6.3. Below the link for adding new hosts is the filter menu. Because each database table can be filtered by different values a separate filter template was added to each list view and is included to the corresponding template. The host table filter menu is displayed in Figure 4:

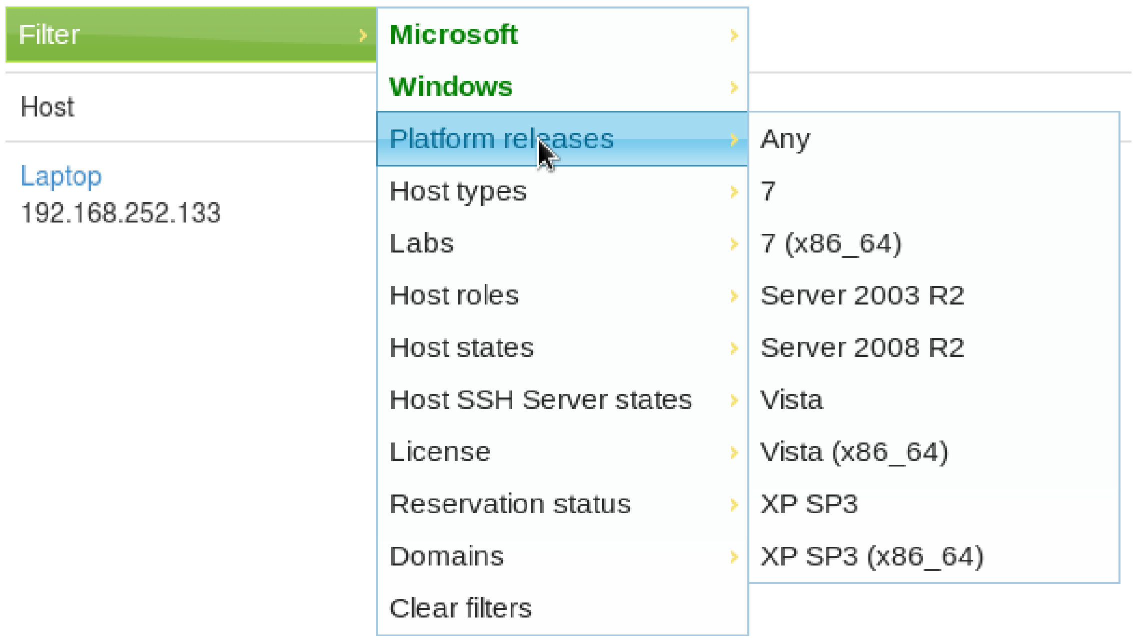


Figure 4. Host table filter menu example

The filter look was implemented using the jQueryUI library widget “Menu”. This widget allows the making of expandable HTML lists as shown in the above example. The filtering itself required custom JS libraries to work. Each select in the filter menu is an unordered list. Every item in that list was given a *name* attribute corresponding to the model field and a *value* attribute denoting the fields’ value according to which filtering is to be done. Clicking an item in the list also added another attribute *selected* to that item to signify the filter selection. After that all lists in the menu were checked for list items with the *selected* attribute and a GET request URL containing all selected name-value pairs was constructed. Then the request was sent to the web server and database objects were filtered as shown in Listing 17 and the user was redirected to a page containing the list of filtered object.

Below the filter menu is a table with all database objects that were received from the database. The HostListView is the only view with additional JS requirements. Functionality for reserving a test host, clearing the reservation and checking the host’s

SSH server status were added to the “Actions” column of the table. The table cell under that column has a calendar icon and a check mark icon. The calendar icon was added for reserving the test host. Clicking that icon shows a calendar with clickable dates. The calendar is another widget from the jQueryUI library. The calendar for host reservation is displayed in Figure 5:

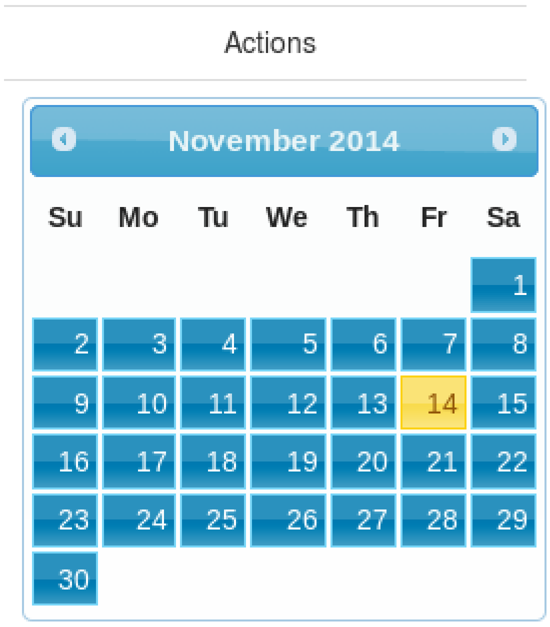


Figure 5. Calendar for host reservations.

Selecting a date until which the host is to be reserved generated a date value that was put into a *hidden* text field. With additional JS code an AJAX POST request was sent to the backend for processing. An example of an AJAX POST request is illustrated in Listing 27:

```
$.ajax({
    type: “POST”.
    url: “/app/host/reserve”,
    data: {“user_id”: (logged_in_user_id),
           “date”: (date)
           “host_id”: (host_id)
           },
    dataType: “json”
}).error(function (jqXHR, status, error) {
            toastr.error(error);
})
```

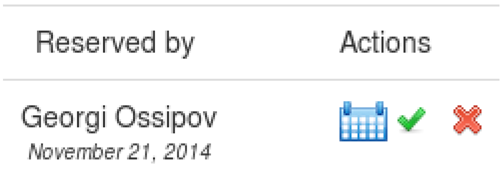
Listing 27. AJAX POST request call

If the AJAX call is successful there is no notification about that success and the user can see from the page that the host has been reserved. If for some reason the backend cannot process the request an error is raised in a *toastr* style “balloon”. The URL to which AJAX makes the call points to a FBV with modifying host’s reservation data functionality. A simplified example of that function id illustrated in Listing 28:

```
def reserve_host(request):
    context = dict()

    if request.method == 'POST':
        if request.is_ajax():

            user_id = int(request.POST.get('user_id', None))
            host_id = int(request.POST.get('host_id', None))
            date = request.POST.get('date', None)

            day = int(date.split('-')[2])
            month = int(date.split('-')[1])
            year = int(date.split('-')[0])

            if user_id and host_id and date:

                host = Host.objects.get(pk=host_id)

                new_date = datetime.date(year, month, day)

                host.th_reserved_by =\
                                User.objects.get(pk=user_id)

                host.th_reserved_until = new_date

                req_host.save()

    return context
```

Listing 28. Host reservation function in python code

Once the host is successfully reserved an icon of a red cross appears in the “Actions” column. An example of a reserved host is displayed in Figure 6:



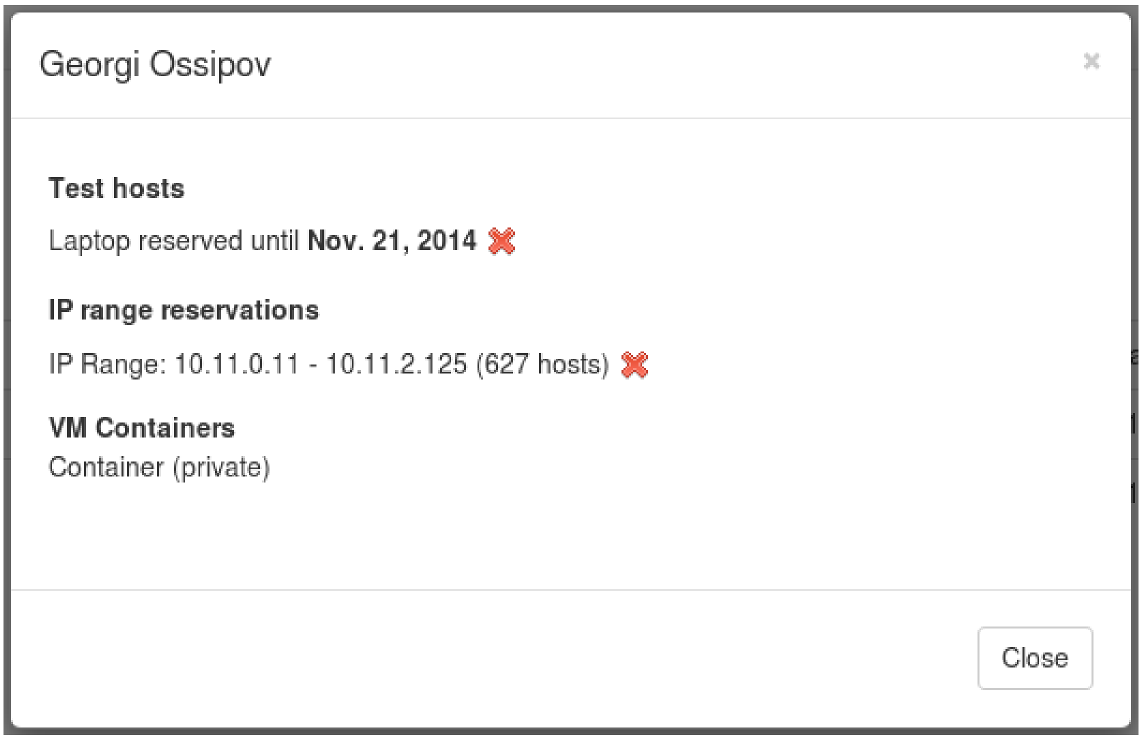
Reserved by	Actions
Georgi Ossipov *November 21, 2014*	📅 ✓ ✗

Figure 6. Example of a reserved host in the front-end.

Clicking the icon will make another AJAX call to a function similar to the one in Listing 28. Only this function will set the *th_reserved_by* and *th_reserved_until* fealds to None which means that the host is available.

Another icon available in the "Actions" column is the green check icon. Clicking this icon will make another AJAX call to a URL pointing to the *get_host_server* function described in Listing 24. If the host's SSH server is running the function will return a string containing the server's information and it will be displayed in the "SSH Server" column.

When working on a number of test cases that require several hosts to be used at the same time a user might want to check which hosts he/she has reserved. This information was added to the users detail view. An example of that view is displayed in Figure 7:



Georgi Ossipov

Test hosts

Laptop reserved until **Nov. 21, 2014**

IP range reservations

IP Range: 10.11.0.11 - 10.11.2.125 (627 hosts)

VM Containers

Container (private)

Figure 7. User details view example

This view contains a list of hosts and a list of IP ranges reserved for the user and any virtual machine containers that were added by the user and marked as private.

Having a host reserved and not needing it anymore might prevent other users from using it. Searching for all reserved hosts from the host list view at times is time consuming and distracting from work. To make that job easier a host list was added to the user's details view. From that view the user can check all valid reservations and if they are no longer needed they can be easily removed making the host "Available". The red cross icon in the view has the same functionality as the same icon in the host list view.

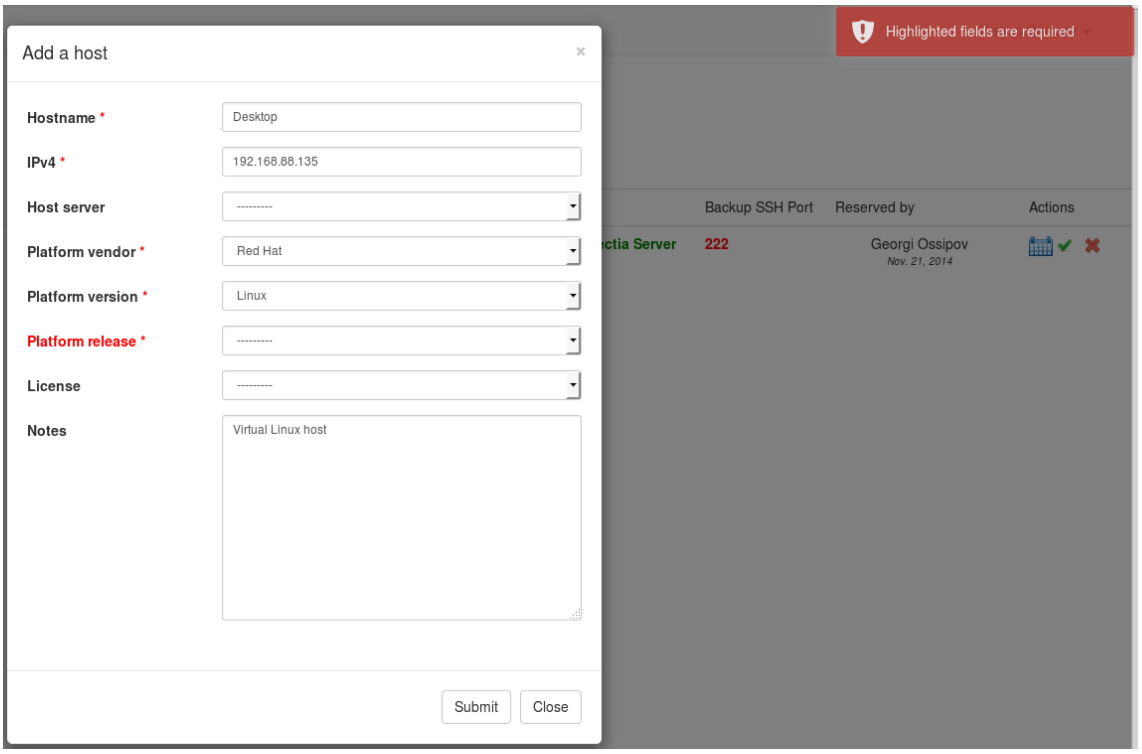
6.3. Form Processing

In the management system all forms are based on models. When a form is submitted first the backend checks if all required fields are filled. This presents unnecessary client server communication for simple tasks. In order to check the required fields' values additional JS files were added to the project. Because all forms are model based the *required* attribute of the form field cannot be set to *False*. A model form field can be left empty if the corresponding model field accepts *blank* or *null* values. Otherwise the field is required. However the attribute can be referenced in templates. To make the form's required fields more visible to the user a red asterisk was added to the fields that cannot be left empty. This is done using a *mixin* that initialized the form before rendering. An example of this mixin is illustrated in listing 29:

```
class RequiredFormFieldsMixin(object):

    def __init__(self, *args, **kwargs):

        super(RequiredFormFieldsMixin, self).\
                                 __init__(*args, **kwargs)

        for field in self.fields:
            if self.fields[field].required:
                self.fields[field].widget.\
                                 attrs['class'] = 'required'
```

Listing 29. Required form field mixin

This mixin will iterate through all form fields of the given form class and add a required class to the fields' input tag for all required fields. When submitting a form the JS code will iterate through all fields with the class *required* and if the field is empty it will get highlighted and an error message will appear. An example of submitting a form with an empty required field is displayed in Figure 8:


Add a host

Hostname *	Desktop
IPv4 *	192.168.88.135
Host server	---------
Platform vendor *	Red Hat
Platform version *	Linux
Platform release *	---------
License	---------
Notes	Virtual Linux host

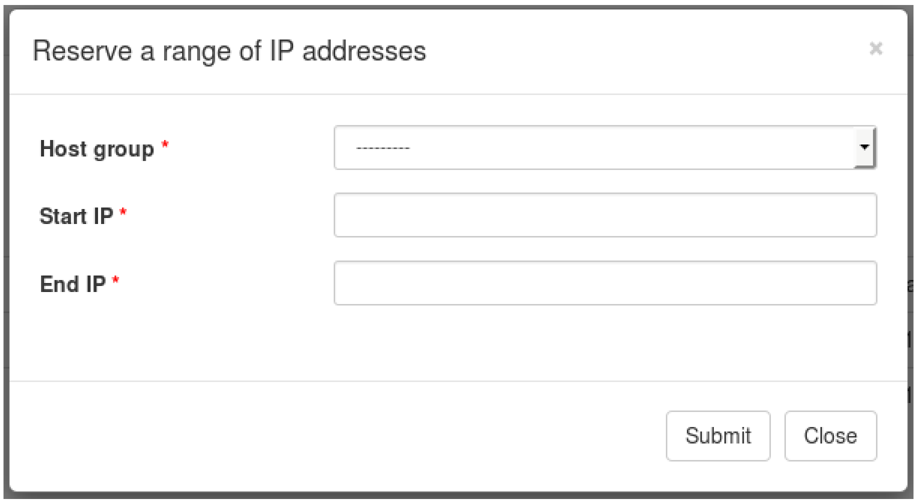
Highlighted fields are required

	Backup SSH Port	Reserved by	Actions
ctia Server	222	Georgi Ossipov Nov. 21, 2014	

Figure 8. Submitting a form with an empty required field

Form errors are implemented with an open source JS library *toastr*. This library allows the developer to display error messages in “balloons” as seen in the figure above. The color of the message “balloon” depends on the status of the message. Other message statuses include success, info and warning.

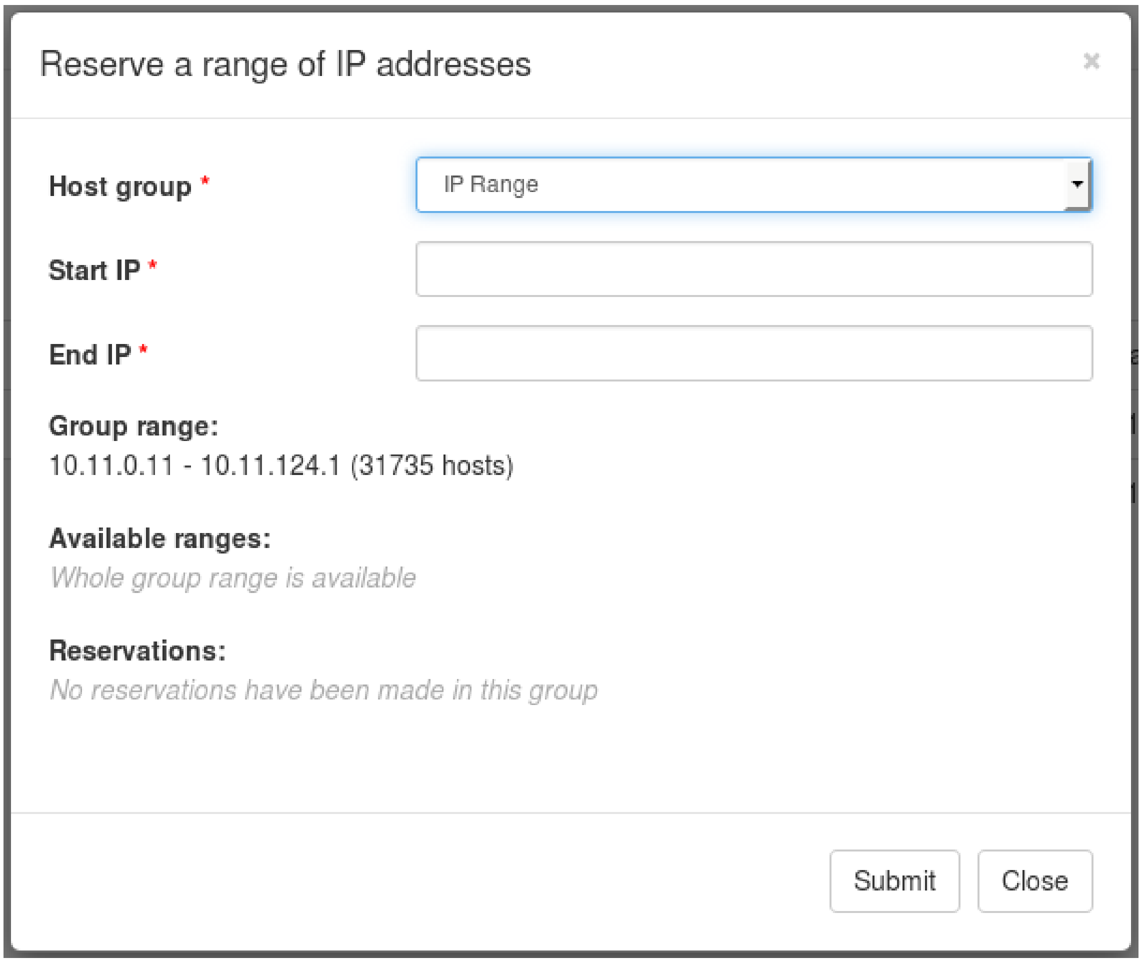
The most difficult form to implement was the IP range reservation form because of additional validators that were implemented. The initial range reservation form is displayed in Figure 9:


Reserve a range of IP addresses

Host group *	---------
Start IP *	
End IP *	

Figure 9. Initial IP range reservation form

If a user needs to reserve a certain range of hosts first a host group needs to be selected from the drop down lost. After the host group is selected an AJAX call is made to the server to get information about the available IP ranges in that group. As illustrated in Figure 10:


Reserve a range of IP addresses

Host group * IP Range

Start IP *

End IP *

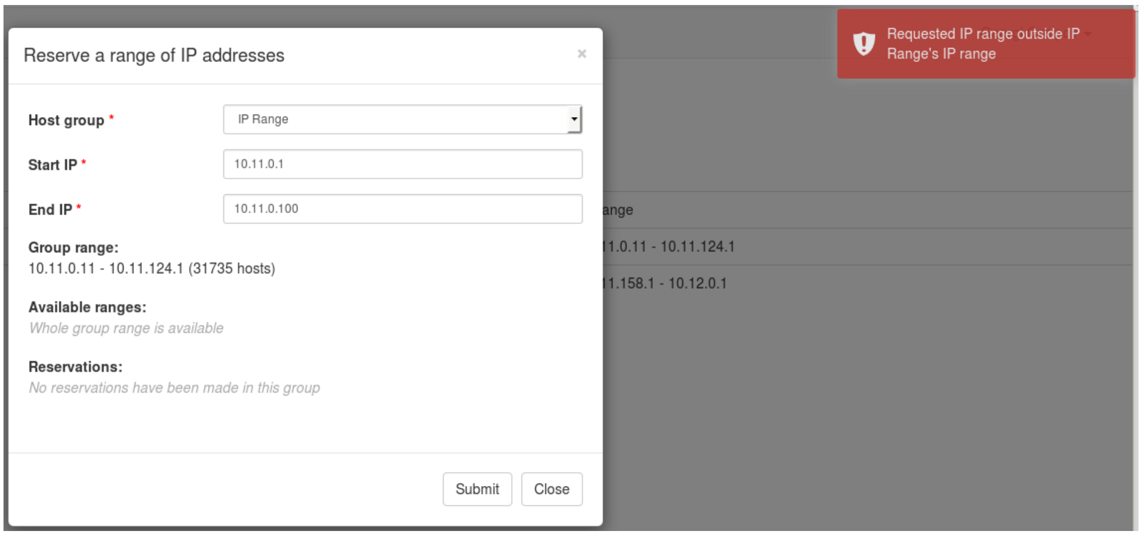
Group range:
10.11.0.11 - 10.11.124.1 (31735 hosts)

Available ranges:
Whole group range is available

Reservations:
No reservations have been made in this group

Figure 10. IP range reservation for for a selected range

As seen from the example above no reservations have been made in that IP range. To make a reservation the user needs to specify the start and end IP addresses. After specifying them the back end will check if the requested IP addresses belong to that group. If the user accidentally entered wrong IPs a *toastr* error message will be displayed. As illustrated in figure 11:


Reserve a range of IP addresses

Host group *	IP Range
Start IP *	10.11.0.1
End IP *	10.11.0.100

Group range:
10.11.0.11 - 10.11.124.1 (31735 hosts)

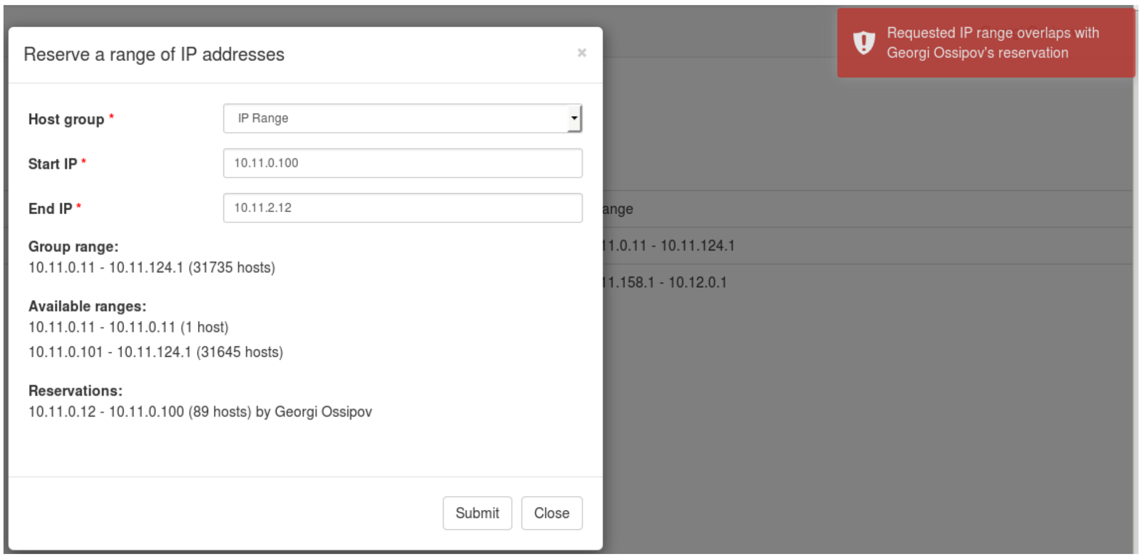
Available ranges:
Whole group range is available

Reservations:
No reservations have been made in this group

Requested IP range outside IP Range's IP range

Figure 11. Error message when reserving an IP range

When a user selects a host group from the drop down list an AJAX call is made to the back end to get the available IP ranges for the selected group. The details also include the overall number of hosts in that group and a number of reserved and available hosts. If there are already some ranges reserved submitting a form will also check if the currently requested range overlaps with any of the existing reservations. An example of these details is displayed in Figure 12:


Reserve a range of IP addresses

Host group *	IP Range
Start IP *	10.11.0.100
End IP *	10.11.2.12

Group range:
10.11.0.11 - 10.11.124.1 (31735 hosts)

Available ranges:
10.11.0.11 - 10.11.0.11 (1 host)
10.11.0.101 - 10.11.124.1 (31645 hosts)

Reservations:
10.11.0.12 - 10.11.0.100 (89 hosts) by Georgi Ossipov

Requested IP range overlaps with Georgi Ossipov's reservation

Figure 12. Host group’s reservation details and error meddage for overlapping reservations

This chapter focused on describing the front end of the management system. Examples and explanations of the main front end functionality were illustrated in Figures 1-12.

7. Conclusion and Discussion

Although the management system is ready for deployment, it can still be improved. For example the framework has additional applications that can be used for developing an Application Programming Interface (API). An API allows the user to interact with an application through the Command Line Interface (CLI).

Using the framework’s built-in functionality for rendering templates requires more memory from the back-end than it would, for example, from the web browser. In order not to use the server for rendering purposes all templates can also be rendered in the web browser. This will transfer a percentage of the work required to the end-user’s computer. Such functionality can be implemented by developing only the API functionality. In order to lessen memory consumption of the server the front-end functionality can be developed using an open source JS library *angular.js*.

Angular is another front-end framework written in JavaScript. It allows the web browser to interact with the web server with less code and more efficient memory usage. In order to implement the front-end, AJAX calls are made to the back-end and the received data is processed by the web browser. Processing database objects in the front-end is possible if they are represented in *json* format. Python has a number of libraries to handle *json* type variables. Developing a proper API will take database entries and convert them into *json* that can be processed by the web browser. This will make the memory usage in the back-end more efficient, because it will not have to render templates.

Overall the project can be improved in many ways. However for the moment it is efficient enough to be taken into use by the company. In the future if the management system needs to be renewed, another way of implementing it could be used. This will let the developer of the system learn new tools for creating web applications.

Conclusion

The goal of this project was to develop an online application for managing a company's network infrastructure to a certain extent. The project was completed using the Django web application framework and a number of front-end frameworks.

In the beginning the application was planned for reserving testing hosts for software testing. During the development additional functionality such as license management, configured domain overview and available hardware monitoring were added. However the main functionality was focused on testing host management.

The developed application allows the user to connect to a remote host in the same network and get information regarding the installed SSH server on that host. This feature required the basic understanding remote connectivity and how it can be implemented using Python programming language. Another feature that required more logical thinking is the IP range management. This included avoiding IP address overlapping in different IP ranges and IP range reservations.

Although the application gives an overview of the internal company network it lacks the functionality for applying network configurations to remote hosts or creating entries on the DNS servers. Static IP addresses and host names need to be configured manually on each testing host and added to the DNS server. This feature can be added to the developed application; however this will require further studies of domain name systems and how they can be interacted with using Python programming language.

Further improvements may also include development of RESTful APIs. This will make interaction with a web application possible not only via a web browser, but also using the command line interface.

References

1. Milovanovic I. Python Data Visualization Cookbook [e-book]. Birmingham, UK, Packt Publishing; 2013.

2. Chacon S. Pro Git [e-book]. New York, NY, Apress; 2009

3. Greenfield D., Roy A. M. Two Scoops of Django [e-book]. Cartwheel Web; 2013.

4. Obe R. O., Hsu L. S. PostgreSQL: Up and Running [e-book]. Sebastopol, CA, O’Reilly Media, Inc.; 2012.

5. Lutz M. Python Pocket Reference [e-book]. Sebastopol, CA, O’Reilly Media, Inc.; 2014.

6. Dauzon S. Django Essentials [e-book]. Birmingham, UK, Packt Publishing; 2014.

7. Lutz M. Programming Python [e-book]. Sebastopol, CA, O’Reilly Media, Inc.; 2011.

8. Supervisor: A Process Control System [online]. Agendaless Consulting and Contributors; Nov 12, 2014.
   URL: http://supervisord.org/. Accessed Nov 14, 2014.