UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

The system for monitoring the status of servers and notifying users of an excessive use of system resources

Author:
Yevhenii MOLODTSOV

Supervisor:
Oleg FARENYUK

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Department of Computer Sciences
Faculty of Applied Sciences

Lviv 2021

Declaration of Authorship

I, Yevhenii MOLODTSOV, declare that this thesis titled, “The system for monitoring the status of servers and notifying users of an excessive use of system resources” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"Any sufficiently advanced technology is indistinguishable from magic."

Arthur C. Clarke

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Bachelor of Science

The system for monitoring the status of servers and notifying users of an excessive use of system resources

by Yevhenii MOLODTSOV

Abstract

Nowadays servers have become an important part of the IT infrastructure and the need of monitoring their health is growing. The consequences of not tracking the server resources could be different - from losing money to losing customers. Sometimes even people's lives are dependent on the stability of the server's infrastructure. The first step of controlling the health of a server is to see and analyze its key metrics, such as CPU, RAM, and HDD utilization. My program provides an easy way of monitoring the system resources and being alerted in case some of them are higher than was expected.

Acknowledgements

I would like to express my sincere gratitude to Oleg Farenyuk for his assistance at every stage of the research project.
And I am deeply grateful to Serghei Burca and Vasilii Burca for their insightful comments and suggestions.

Contents

List of Figures

List of Abbreviations

API	**A**pplication **P**rogramming **I**nterface
AWS	**A**mazon **W**eb **S**ervices
CI/CD	**C**ontinuous **I**ntegration/**C**ontinuous **D**eployment
CPU	**C**entral **P**rocessing **U**nit
CRUD	**C**reate **R**ead **U**pdate **D**elete
DDoS	**D**istribute **D**enial-**o**f-**S**ervice
DRF	**D**jango **R**est **F**ramework
DRY	**D**onť **R**epeat **Y**ourself principle
HTTP	**H**yper**T**ext **T**ransfer **P**rotocol
IT	**I**nformation **T**echologies
MVC	**M**odel **V**iew **C**ontroller pattern
MVP	**M**inimum **V**iable **P**roduct
MVT	**M**odel **V**iew **T**emplate pattern
ORM	**O**bject **R**elational **M**apping
PC	**P**ersonal **C**omputer
REST	**RE**presentational **S**tate **T**ransfer
SSL	**S**ecure **S**ocket **L**ayer
URL	**U**niform **R**esource **L**ocator

Dedicated to my parents who supported me in getting the bachelor's degree and my friends thanks to whom I did not go crazy while writing this thesis…

Chapter 1

Introduction

A server is a computer that is meant to be a dedicated service provider [*Windows Server Administration Fundamentals* 2011]. In general, every time the computer shares data with another device, it could be considered as a server, while the device is called a client. With the widespread use of the Internet servers became the heart of the whole IT industry. Worldwide end-user spending on public cloud services is forecast to grow 18.4% in 2021 to total $304.9 billion, up from $257.5 billion in 2020 [Gartner, 2017].

Nowadays the access to data has become the main reason for using PCs and laptops. Most existing businesses try to provide their solutions and attract some new customers through the web [Lesonsky, 2017]. Almost all companies want to automate some processes and this wish leads them to use cloud platforms. There is a wide range of spheres where the web could be used:

1. Social platforms

   Nowadays a big part of communication is going online, especially during the corona pandemic. All the media platforms like Facebook, Instagram, Twitter, and others have lots of users and they need advanced cloud solutions to provide a full-time fast access to their services. All the video and audio meeting platforms are also included in this list.

2. Sells sphere

   The main point of most businesses is to sell something to potential clients. Nowadays there are lots of selling platforms on the web. From Amazon to Rozetka, to Aliexpress. All of those solutions need a huge and scalable cloud platform to maintain all the clients and their data, as well as the products themselves. It is also important to bring a high level of security to such systems, so the individual preferences of users or their payments data are not shared anywhere else.

3. Entertainment sphere

   There are some video streaming services like YouTube, Netflix, or Twitch. All of them need highly optimized web solutions to deal with all the video streaming as well as fast hardware and advanced load balancing solutions to handle all their millions of clients.

4. Automating existing business solutions

   There are lots of businesses that deal with physical operations. For example, the editions and covers management system. There's lots of management underneath -= the arrivals of new editions, distributing those across the stores, printing editions, and covers, taking into account that each publisher wants some management system to be able to publish the covers and change them in different editions of the same title, sometimes even dependent on the region. All these processes require some automated solution with a strong cloud system to be able to maintain all the titles data. It's also worth mentioning that such a solution needs a flexible user management system because there are lots of managers with different roles and responsibilities – from content library managers and publishers to the admins of the whole system.

Of course, the list above is not full and it could be expanded furthermore. But the main point here is that in our modern world lots of business, social, and entertainment processes are presented on the web. For all of them, there is a need to provide some customizable cloud solutions as well as highly performed balancers to maintain all the clients and their traffic.

Chapter 2

Problem

When the company grows, the number of its clients as well as the amount of processed data increases. Most companies today are client-oriented [Wirtz and Daiser, 2018]. Even if a company serves some business needs, most of the time it needs the web resources to be able to sell its solutions. The efficiency and stability of systems are now a high priority for the web industry. There is such a notion as server overloading and it can be caused by different reasons:

1. Malware or viruses

   This one is about server security, but it can be the reason for slower behavior as was expected.

2. DDoS attacks

   Those are also mainly about the server setup and security and they are not relevant to the issue my application solves, but those can significantly slow the application.

3. Hardware

   There are some components in the system that can make an influence on its working speed. The most important of them are hard drive, memory and CPU, and their speed. There are also some less significant ones like virtual memory or bus speeds. If the hardware is cheap and slow it is harder to build a responsive system.

4. System resources

   Those are about what amount of memory, hard drive space, and CPUs the system needs. If there are not enough of them the system may longer process the requests or go down at all.

All of the reasons described are valuable and should be considered when building a web application. But the last one could be improved from the software perspective. There is a need to monitor system resources to know when it's needed to start scaling them. In general, there are different type of dealing with lack of resources:

1. Software optimizations

   These are about optimizing database queries, making asynchronous operations when the CPU time is mostly spent on the input-output. This step should be considered as the first one when dealing with overloading.

2. Hardware scaling

   This is about increasing the number of hardware resources on the instance. It is a pretty straightforward and dirty solution.

3. Server Load Balancing [*What Is Load Balancing?* 2018]

   This is the most relevant and elegant solution, but it requires related knowledge to build the system properly. It could be software or hardware balancing solutions and the last one is harder to implement and more expensive to maintain. The software implementation is a layer that is placed between the server and the clients and it is responsible to traffic all the clients' requests to the server instances. This solution is flexible and easy to scale.
   Also, some efficient balancers use dynamic load balancing algorithms to scale the system dynamically. For example, when there is a web store and it comes to holiday sales, the amount of traffic could be increased significantly. In such a case there is no need to pay for a large number of resources during all the year and here comes the dynamic load balancer. It will add the needed instances when the traffic increases.

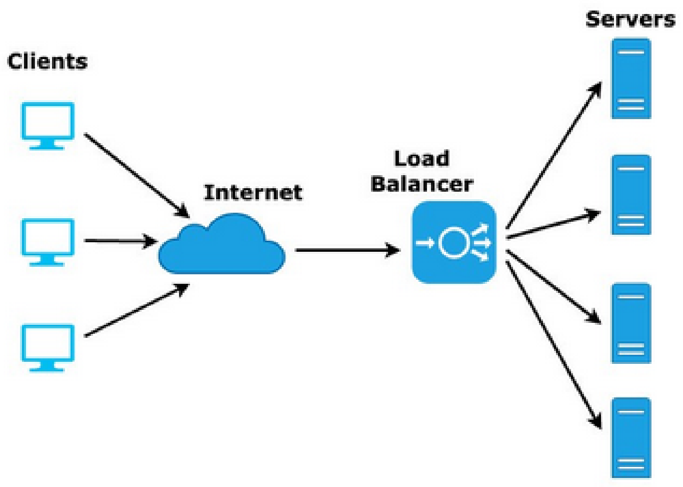


FIGURE 2.1: Load Balancer Layer

The last solution is elegant but it is also pretty expensive and there is no need to build such a system for small businesses. Nevertheless, it is still important to monitor the number of resources consumed to prevent the server from being overloaded. Even if the business decides not to build the load balancer, it still needs to be aware of resource usage to be able to add some instances or hardware manually.

Chapter 3

Existing solutions

As it was mentioned before, there is a need to monitor the server's resources. There are plenty of tools that can help developers and DevOps engineers to see the server's metrics. The most popular ones are usually provided by hosting platforms, such as AWS or Google Cloud. Here are the examples with their benefits:

1. AWS platform [*Amazon CloudWatch*]

   CloudWatch is the observability and monitor service that provides needed data and some insights to monitor your applications, respond to system-wide performance changes, optimize resource utilization, and get a unified view of operational health. It collects monitoring and operational data in the form of logs, metrics, and events, providing you with a unified view of AWS resources, applications, and services that run on AWS and on-premises servers. It can be used to detect anomalous behavior in your environments, set alarms, visualize logs and metrics side by side, take automated actions, troubleshoot issues, and discover insights to keep your applications running smoothly. Amazon CloudWatch is a flexible tool with such advantages:

   (a) Observability on a single platform across applications and infrastructure
   When the application is large it becomes a mess to analyze all the metrics and logs it collects, because usually, it has lots of instances and environments. CloudWatch is a single place where all such information is considered.

   (b) The easiest way to collect metrics
   Cloudwatch integrates with more than 70 services provided by AWS, such as Amazon EC2, DynamoDB, S3 Storage, and others. It also could be used in hybrid cloud architecture Agent or API to monitor resources.

   (c) Get operational visibility and insight
   Amazon provides a great granulated view of all the system resources, as well as up to 15 months to store the collected data. All the charts and models are highly customizable.

2. Google platform [*Cloud Monitoring*]

   Cloud monitoring can collect data from Google Cloud, AWS instances, and other platforms. Using the integration with BindPlane service it is possible to collect data from over 150 common application components and different cloud systems. The Cloud monitoring has the following features:

   (a) SLO monitoring
   Automatically infer or custom define service-level objectives (SLOs) for applications and get alerted when SLO violations occur.

(b) Group and cluster support

Define relationships based on resource names, tags, security groups, regions, accounts, and other criteria. Use those relationships to create targeted dashboards and topology-aware alerting policies.

(c) Alerting

Configure alerting policies to notify you when events occur or a particular system or custom metrics violate rules that you define. Use multiple conditions to define complex alerting rules. Receive notifications via email, SMS, Slack, PagerDuty, and more.

The important part of system monitoring is presenting the data and notifying the end-user when something is going wrong and some thresholds are reached. And both solutions are doing well with it. They both are also highly customizable and support integrations with third parties.

Chapter 4

My solution

The main purpose of the whole program is to bring the users control over the server instances they have. It is possible to add as many instances as they wish and all of them will be checked according to the needed threshold percentage.

It collects the data about some basic metrics like CPU, RAM, HDD, Network usage, etc. The users can enable or disable them as they wish. The program will run the background task to check the thresholds and notify a user if some of them are reached. Of course, there are multiple ways of notifying the user: front-end alert, logs on the server instance, chat-bots - all of them are supported and can be set up.

I created a program that is easy to configure and use. It also scales well and allows the end-users to have as flexible a setup as they wish. It is worth mentioning that for now, it is only the back-end application with the Admin page to basic data management and REST API to integrate the front-end or other services.

So, what are the advantages of my program comparing to the existing solutions described above?

1. Simplicity and cost on the start

   The described services are good and provide a great monitoring solution, but they're too heavy for small instances. The complexity of configuring AWS CloudWatch or Google Cloud Monitoring is high and it requires hiring people that know how to do this. This will cost additional investments. My program is more lightweight and easy to start using out of the box.

2. The absence of necessity of using some Cloud Provider

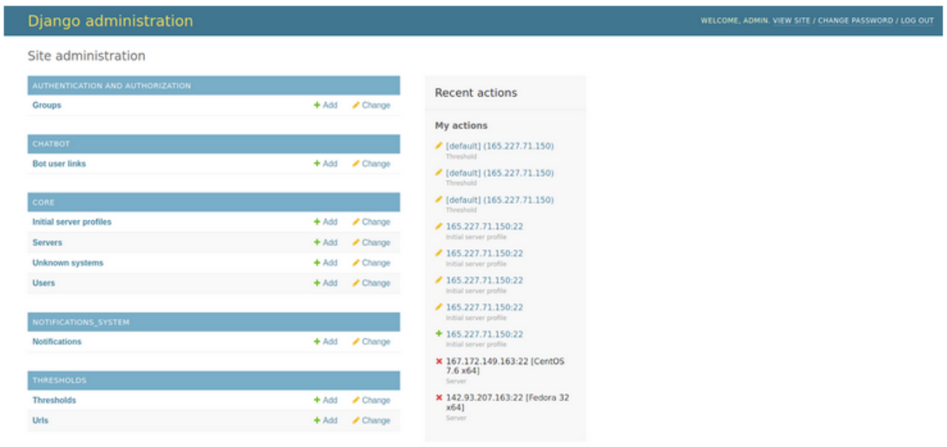
   These tools are great when it comes to integration with their Cloud Providing systems and they become a huge mess when trying to integrate them with your custom servers instances that are served directly by you. My program allows the users to connect any instance that has ssh (or just basic) authentication.

Chapter 5

How it works

5.0.1 User interface

The main strategy was to create the user interface (the flow of user interaction) as friendly as possible. However, the Django Admin takes all the responsibility for user interaction.
The main Admin page allows users to see what type of resources they have:



[default] (165.227.71.150)
Threshold
[default] (165.227.71.150)
Threshold
[default] (165.227.71.150)
Threshold
165.227.71.150:22
Initial server profile
165.227.71.150:22
Initial server profile
165.227.71.150:22
Initial server profile
165.227.71.150:22
Initial server profile
165.227.71.150:22
Initial server profile
167.172.149.163:22 [CentOS 7.6 x64]
Server
142.93.207.163:22 [Fedora 32 x64]
Server

FIGURE 5.1: Django Admin Page

It is also possible to see all the users of the system if you are the admin, as well as manage the groups and users' permissions.
With this admin it is possible to add some of the server instances you have:

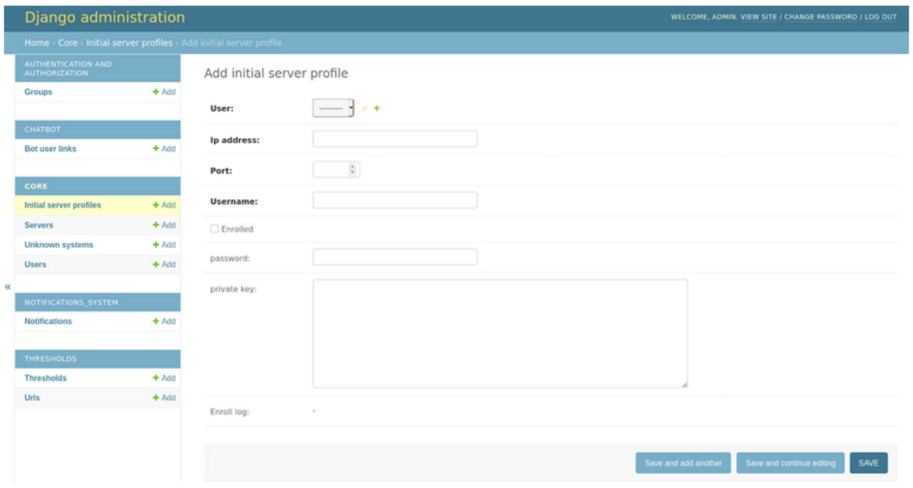
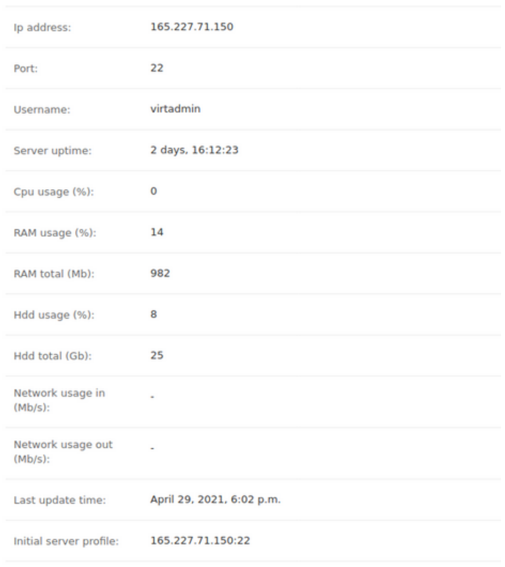


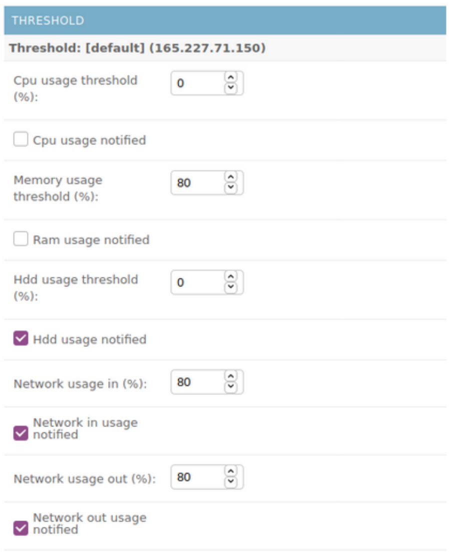
FIGURE 5.2: Admin Server Instance

And then see all their thresholds as well as setting up the needed level of thresholds (for being notified after they're reached)



Ip address:	165.227.71.150
Port:	22
Username:	virtadmin
Server uptime:	2 days, 16:12:23
Cpu usage (%):	0
RAM usage (%):	14
RAM total (Mb):	982
Hdd usage (%):	8
Hdd total (Gb):	25
Network usage in (Mb/s):	-
Network usage out (Mb/s):	-
Last update time:	April 29, 2021, 6:02 p.m.
Initial server profile:	165.227.71.150:22

FIGURE 5.3: Detailed Server Instance



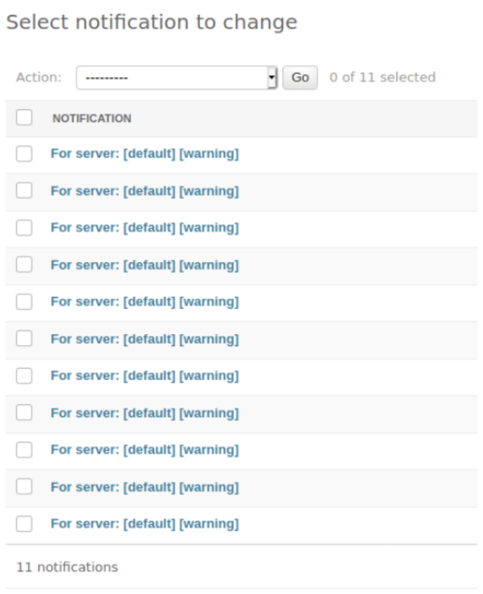
THRESHOLD

Threshold: [default] (165.227.71.150)

Cpu usage threshold (%):	0
☐ Cpu usage notified	
Memory usage threshold (%):	80
☐ Ram usage notified	
Hdd usage threshold (%):	0
✓ Hdd usage notified	
Network usage in (%):	80
✓ Network in usage notified	
Network usage out (%):	80
✓ Network out usage notified	

FIGURE 5.4: Detailed Threshold Instance

After reaching the thresholds server creates the notification object in the database.



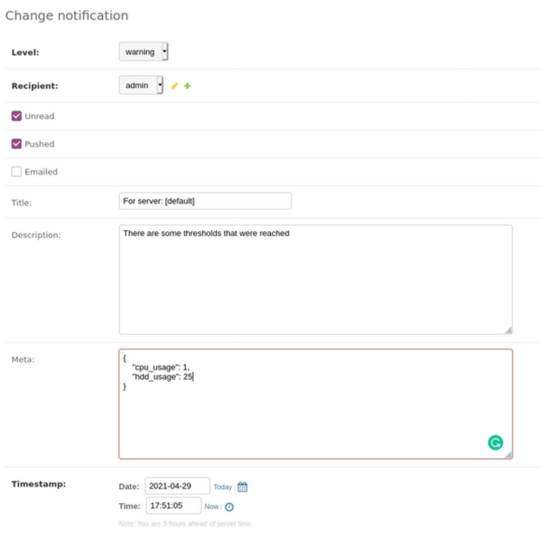
Select notification to change

Action: --------- Go 0 of 11 selected

NOTIFICATION
For server: [default] [warning]
For server: [default] [warning]
For server: [default] [warning]
For server: [default] [warning]
For server: [default] [warning]
For server: [default] [warning]
For server: [default] [warning]
For server: [default] [warning]
For server: [default] [warning]
For server: [default] [warning]
For server: [default] [warning]

11 notifications

FIGURE 5.5: Notifications



Change notification

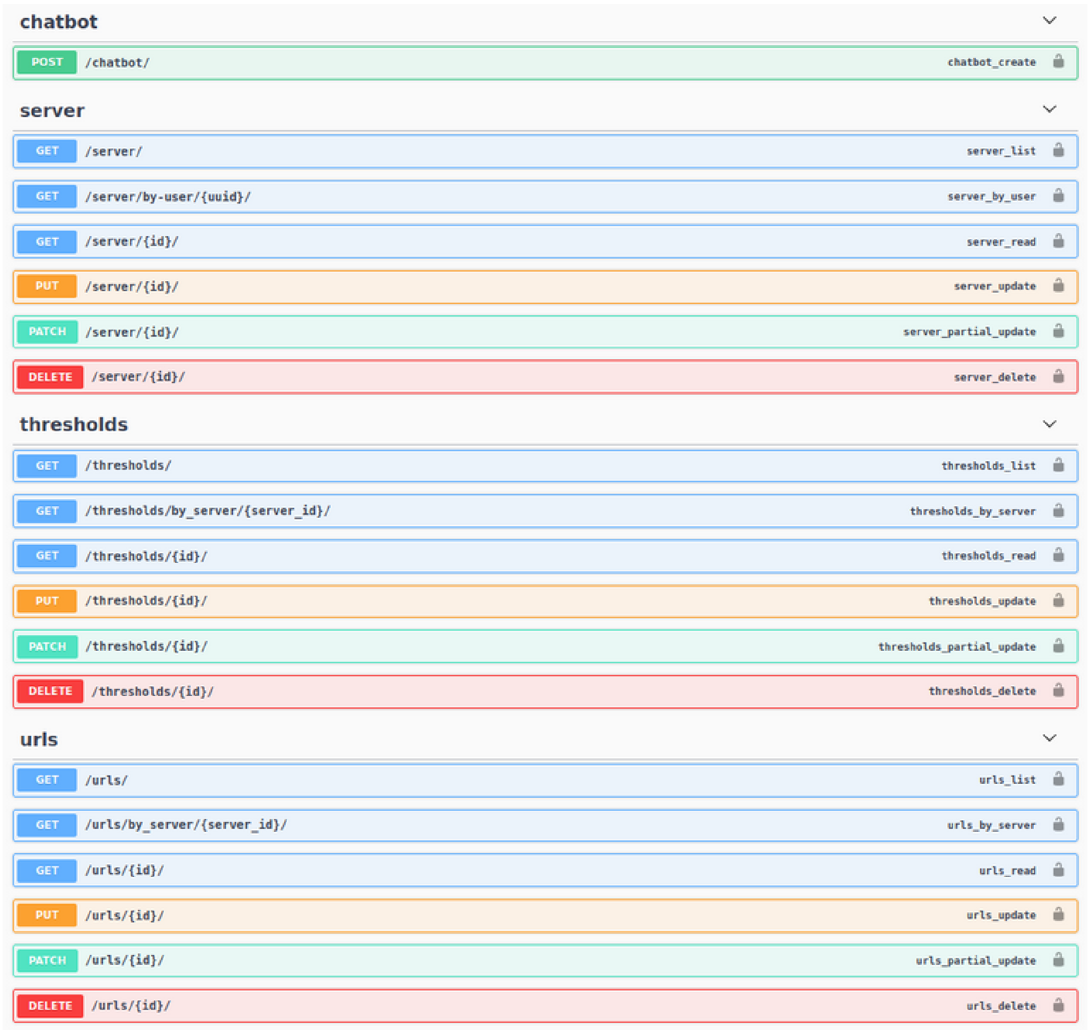
Level:	warning
Recipient:	admin
✓ Unread	
✓ Pushed	
☐ Emailed	
Title:	For server: [default]
Description:	There are some thresholds that were reached
Meta:	{ "cpu_usage": 1, "hdd_usage": 25 }
Timestamp:	Date: 2021-04-29 Today; Time: 17:51:05 Now

Note: You are 3 hours ahead of server time.

FIGURE 5.6: Detailed Notification

Later those notification objects are used to send the alerts to the frontend, emails. There is also a possibility to connect both Facebook and Telegram chatbots to this system to receive some alerts.

It could be seen from my screenshots that the main interface is the Admin Panel. But as I mentioned before, there is a REST API interface to integrate the frontend:



chatbot

```
POST    /chatbot/                                   chatbot_create
```

server

```
GET     /server/                                    server_list
GET     /server/by-user/{uuid}/                     server_by_user
GET     /server/{id}/                               server_read
PUT     /server/{id}/                               server_update
PATCH   /server/{id}/                               server_partial_update
DELETE  /server/{id}/                               server_delete
```

thresholds

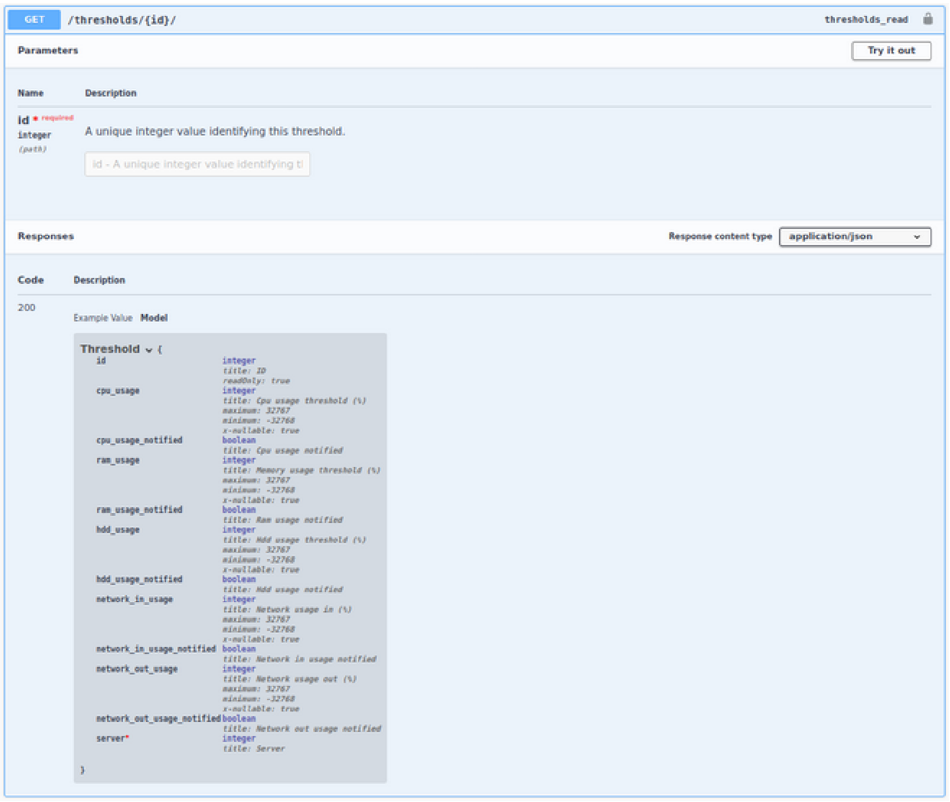
```
GET     /thresholds/                                thresholds_list
GET     /thresholds/by_server/{server_id}/          thresholds_by_server
GET     /thresholds/{id}/                           thresholds_read
PUT     /thresholds/{id}/                           thresholds_update
PATCH   /thresholds/{id}/                           thresholds_partial_update
DELETE  /thresholds/{id}/                           thresholds_delete
```

urls

```
GET     /urls/                                      urls_list
GET     /urls/by_server/{server_id}/                urls_by_server
GET     /urls/{id}/                                 urls_read
PUT     /urls/{id}/                                 urls_update
PATCH   /urls/{id}/                                 urls_partial_update
DELETE  /urls/{id}/                                 urls_delete
```

FIGURE 5.7: Swagger Schema

The API interface supports all the CRUD operations, so the frontend can build its own CMS based on this backend solution. All the API endpoints are documented and there is a possibility to see all the needed parameters and possible responses for each URL as well as try them in “live mode” to see how the data is returned:



```
GET /thresholds/{id}/
```

Name	Description
id * required integer (path)	A unique integer value identifying this threshold. id - A unique integer value identifying t

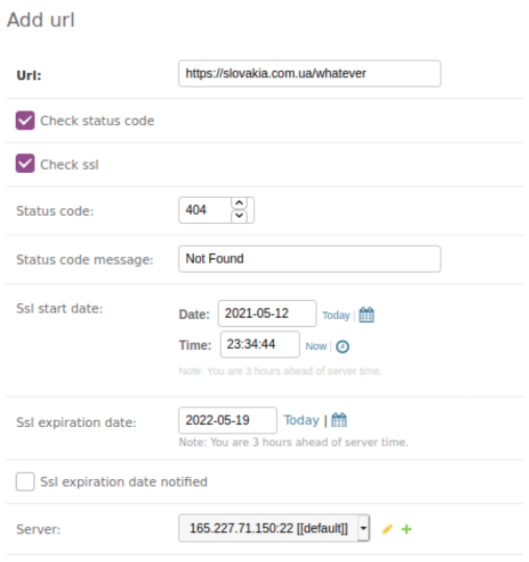
Response content type: application/json

Code	Description
200	Example Value Model

```
Threshold {
   id                          integer
                               title: ID
                               readOnly: true
   cpu_usage                   integer
                               title: Cpu usage threshold (%)
                               maximum: 32767
                               minimum: -32768
                               x-nullable: true
   cpu_usage_notified          boolean
                               title: Cpu usage notified
   ram_usage                   integer
                               title: Memory usage threshold (%)
                               maximum: 32767
                               minimum: -32768
                               x-nullable: true
   ram_usage_notified          boolean
                               title: Ram usage notified
   hdd_usage                   integer
                               title: Hdd usage threshold (%)
                               maximum: 32767
                               minimum: -32768
                               x-nullable: true
   hdd_usage_notified          boolean
                               title: Hdd usage notified
   network_in_usage            integer
                               title: Network usage in (%)
                               maximum: 32767
                               minimum: -32768
                               x-nullable: true
   network_in_usage_notified   boolean
                               title: Network in usage notified
   network_out_usage           integer
                               title: Network usage out (%)
                               maximum: 32767
                               minimum: -32768
                               x-nullable: true
   network_out_usage_notifiedboolean
                               title: Network out usage notified
   server*                     integer
                               title: Server

}
```

FIGURE 5.8: Detailed Swagger Schema

It is also possible to setup some URLs to check their HTTP statuses as well as SSL certificates expiration dates:



Add url

Url:	https://slovakia.com.ua/whatever
✓ Check status code	
✓ Check ssl	
Status code:	404
Status code message:	Not Found
Ssl start date:	**Date:** 2021-05-12 Today | **Time:** 23:34:44 Now | Note: You are 3 hours ahead of server time.
Ssl expiration date:	2022-05-19 Today | Note: You are 3 hours ahead of server time.
☐ Ssl expiration date notified	
Server:	165.227.71.150:22 [[default]]

FIGURE 5.9: Detailed Url Object

5.0.2 User interaction flow

The user interaction flow is pretty simple and has the following steps:

1. Creation of initial server

   User needs to enter the following data:

   - Ip address
   - Port (default for ssh is 22)
   - Username
   - Password or ssh key

   That's it! After submitting this data (via saving on the Admin page or sending the POST request) the server enrollment process will start. The server enrollment process is a bash script and it has the following steps:

   (a) Authenticate the application on the server instance
   (b) Create the application user (virtadmin)
   (c) Create the home directory for this user, so all the monitoring-related data is separated from the instance data
   (d) Put the ssh key on the server-side, so the application can have direct access
   (e) Create a python virtual environment depending on which system is running on the instance, so later it is possible to run the python monitoring script in a separate environment and not installing some needed dependencies globally
   (f) Running the script for the first time
   (g) Checking all the setup and writing the corresponding logs if something went wrong

   No matter if the server enrollment fails or not it is possible to retrieve the logs with all the output during the installation.

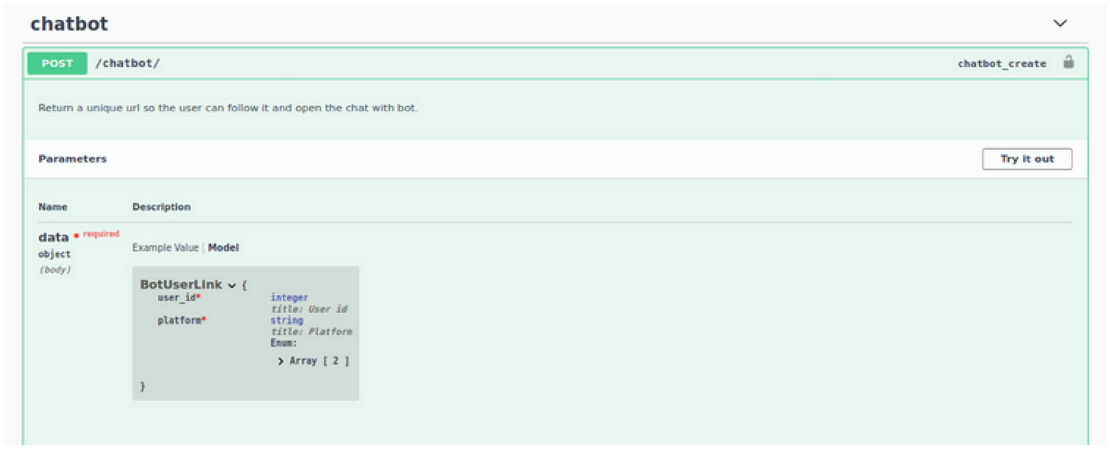
2. Name the server

   During the first stage of server enrollment, the Server object in the database is created. The user needs to name it because the default name is the same for each server and it would be hard to distinguish them later.

3. Customize thresholds

   During the creation of the server, the default thresholds are created. The default value for them is 80 percent. If the users want, they can customize those values according to their needs.

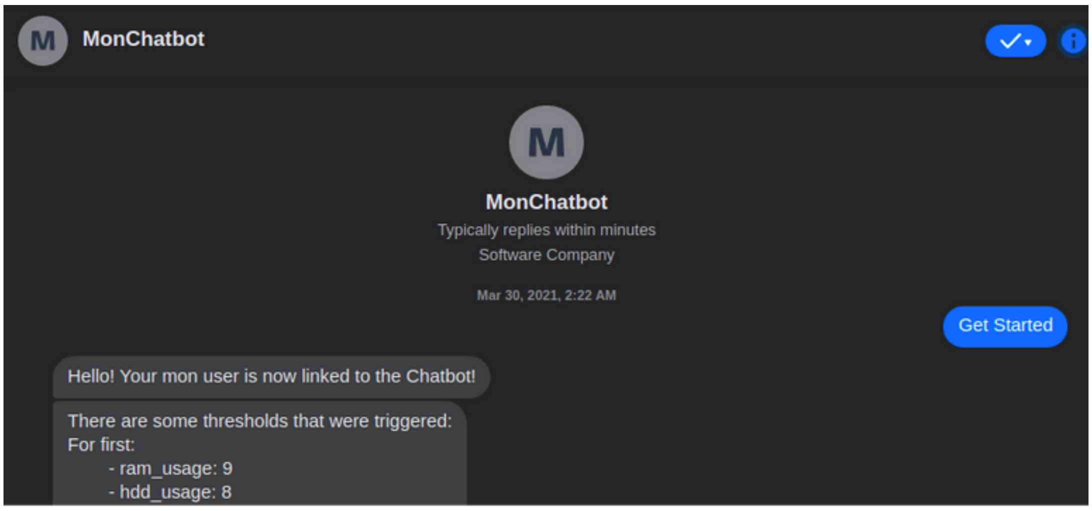
This flow could be slightly changed when writing a separate front-end app. But the flow that was described is the most common use of service.
After the server setup user can also link the chatbot to the system:



```
/chatbot/

Return a unique url so the user can follow it and open the chat with bot.

Parameters

Name             Description
data * required  Example Value | Model
object
(body)           BotUserLink ⌄ {
                    user_id*        integer
                                    title: User id
                    platform*       string
                                    title: Platform
                                    Enum:
                                     > Array [ 2 ]
                 }
```

FIGURE 5.10: Chatbot Swagger Endpoint

Passing the user id and the needed platform (for now only telegram and messenger are supported) the user will get the unique link. Following the link, the user will go to the chat with the bot. This conversation is automatically linked to the user in the system, so later it is possible to receive alerts in this conversation:



Hello! Your mon user is now linked to the Chatbot!

There are some thresholds that were triggered:
For first:
- ram_usage: 9
- hdd_usage: 8

FIGURE 5.11: ChatBot Conversation

Chapter 6

The architecture of my solution

6.1 Monolithic server architecture

There are two main approaches in building the architecture of web software: monolithic and micro-services. Most new companies and startups firstly chose to build their systems with a monolithic architecture [Rud, 2019]. It's understandable because the process of creating a monolith that is responsible for covering all business needs is much easier:

1. Less complex application structure

   All the common services that are used in the web application are easier to maintain. For example, logging, gathering analytics, caching, or perform monitoring – all those features are handled pretty straightforward in the case of monolith because all the sources are located in one place and there is no need to synchronize them across different micro-services.

2. Fewer system resources on the application's infrastructure in the early stages

   Monolithic architecture implies fewer server instances. Of course, with a scale of clients number, a few instances will be needed, so the load balancer can maintain them properly.

3. Faster MVP and proof of concept

   In new companies and especially in startups always exists the need of creating the MVP as fast as possible. In this way, a company can start gathering investments and attract an audience.

4. Less money is spent

   If summarize all the benefits described above, then the main one comes into the game – money. Due to the less complexity of developing and maintaining less money are spent to develop such application.

However, micro-services are considered to be a trend, and most of the well-known organizations like Uber, Netflix, Amazon, eBay, and others have shifted to this approach. It has lots of advantages if compared with monolith:

1. Scalability

   When the project grows, the team also growth and it becomes a mess to work within a large group of people on one source. The much better way is when the work is divided across multiple micro-services, so people can code separately. Also, the scalability touches the software part. It is simpler to scale needed components in the micro-service architecture rather than scaling the whole monolith.

2. Reliability

   The micro-services are independent and communicate through APIs. If this communication is built properly, then a bug in one micro-service has fewer chances to influence the behavior of others if compare with the monolith.

3. Better understanding

   This one is the most subjective because the level of understanding depends on the developer's experience. But in most cases, when a new developer comes to a large project, it is faster to get the micro-service architecture. Moreover, during the on-boarding, newcomers can work on tasks that are not knowledge related to the whole project but corresponds only to the particular micro-service.

4. Possibility to choose technologies for each micro-service

   There could be the case when some parts of code are written in different languages. Moreover, even using one programming language, the framework could be chosen according to the need. For example, the core of the application could be written on Django, some lightweight micro-services on Flask, and some APIs layers on FastAPI due to the support of asynchronous code.

Taking into account all the described above I have chosen the monolithic way of creating my application. It would be much easier for me to maintain it and to build the MVP during a few months.

6.2 Django framework

There are various Python Web frameworks. They are split up into two groups: full-stack and not full-stack ones. The last ones are mostly lightweight and fast. The most popular example of such frameworks is Flask while the FastAPI is the most modern one. It allows developers to build fast and asynchronous APIs. Such frameworks are used when the web application does not have complex business logic. Also, FastAPI is the best choice when the developer builds a service that is a layer between end-user and other services, due to the support of asynchronous operations. It allows handling more requests simultaneously because most of the time the CPU is blocked by Input-Output operations.

On other hand, full-stack frameworks mostly have the highest levels of abstraction. It allows developers to follow the DRY principle. They also come with native support of databases, templates, caching, etc, so they reduce the amount of time and effort to start the application.

The most popular Python full-stack framework is Django. Lots of huge companies use Django for their needs: Spotify, Instagram, YouTube, BitBucket, Dropbox, Pinterest, Mozilla, and others. There are many advantages of using this framework for both large and small applications. But my project is a relatively small one, so here are the main benefits of using Django in this case:

1. The speed of starting a new application

   As it was mentioned before, Django comes with lots of useful functionality out of the box. It has its ORM system, authentication and authorization support, flexible user management, admin panel, and frameworks like DRF to build REST APIs without writing low-level logic, but concentrating on the business part of the solution.

2. MVC (MVT) pattern [George, 2020]

   Patterns are important and widely used in software architecture to build efficient solutions and avoid common mistakes. Model-View-Controller is one of the most famous approaches to building user interfaces. In this pattern, a user uses a Controller to interact with a Model (data) layer and then sees the results of interaction in a View layer.

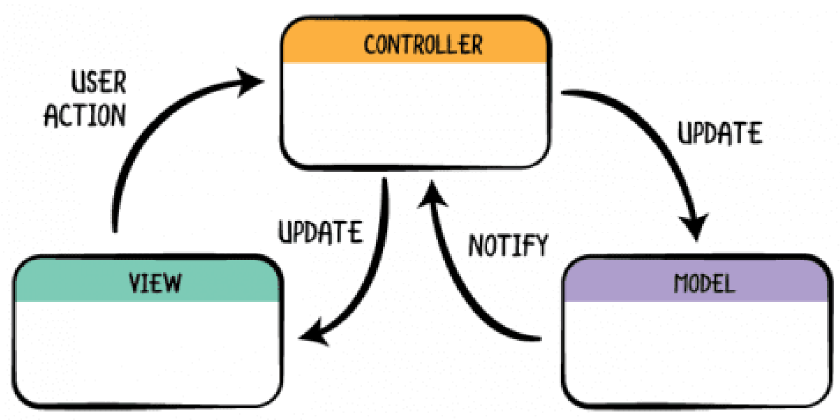


FIGURE 6.1: MVC Pattern [*MVC Architecture - Model, View, Controller*]

   The Django MVC approach is slightly different because the Controller part is mainly handled by the framework by following rules described in the URL config. MVT stands for the Model-View-Template pattern, where the Model layer is the same as in MVC, the Template is the representation layer (View in MVC) and View is the business logic of the application, which is called by the Django Controller, i.e. the logic underneath URL config.

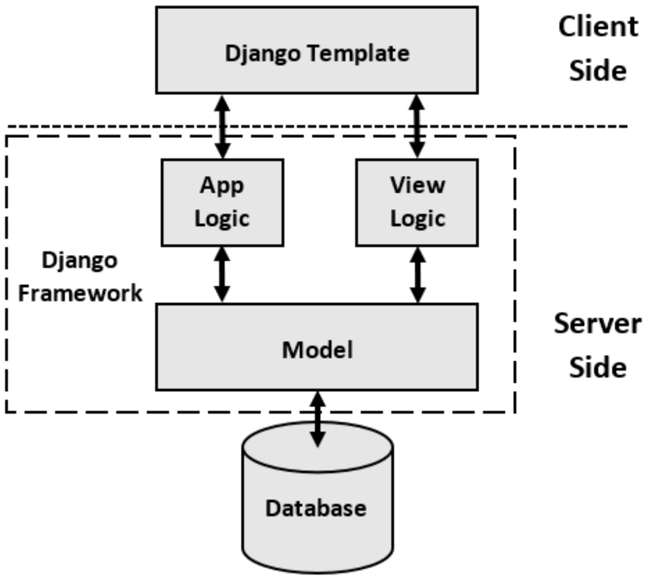


FIGURE 6.2: Django MVT Pattern [George, 2020]

3. Admin interface

   The Django admin interface is a useful panel to work directly with the database. It supports different extensions as well as user customizations. Of course, Django admin is the tool for developers and not the final clients, because it's not user-friendly. Also, it is possible to perform any operations on the database, so the user can destroy some tables or the entire database at all.

Of course, some of the user rights could be limited, but the best solution in the existing product should be to write a separate front-end admin page to manage all the application data. In this way, you as a developer have more control over what operations the user can perform.

Although allowing users to do something on the Django admin is not the best approach and it could be even dangerous, some startups and new projects use it because it is simple, fast, and cheap. As I mentioned before, the Django admin is relatively easy to customize. Using inheritance from the framework's default admin form classes it is possible to override some front-end behavior and even change the existing default layout.

6.3 Django ORM and MySQL database

In Web applications, the user often interacts with data. All the data could be located somewhere in other services and be accessed through APIs – in this case, the application is just a layer between the user and other service providers. But more often a web application has its database to store users, their data, and other data needed for the service. The main responsibility of a website is to give its users some handy interface to access their data. Django framework is suitable to write such database-driven projects. It follows the MVC pattern design, where M-Model is a data layer, handled by the Django ORM system [Holovaty and Moss, 2008].

The initial setup of the database is done from the settings.py using predefined constants. The best practice is to have those variables stored in the env file and load them into the Django app using environment variables. That is how it is done in my project. Using such an approach it is easy to separate database credentials, user secrets, and other settings in different environments (obviously, they should differ in test, staging, or production).

The central notion of Django ORM is a Model. The model is equivalent to Table in SQL. A quick example [*Django Models*]:

```
from django.db import models
class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

The corresponding database query is:

```
CREATE TABLE myapp_person (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(30) NOT NULL
);
```

In my application, there are different tables in each sub-application. As I mentioned before, Django does lots of magic underneath and for this, it requires a predefined project structure. In my application, there are different subprojects, and each of them has its file where the models are defined. It could be seen from the diagram that there are plenty of tables:

1. User
   This table is inherited from the AbstractUser. This way I can customize the Django representation user as I wish. In this case, I could avoid this complexity because I don't need some custom fields in the User model. But according to Django best practices, it's a nice approach to inherit from the AbstractUser at the very beginning of a project and run first migrations with it. This way later, during the development, if any changes or additions to this model are needed – they are easy to implement. If the developer chooses not to implement this approach, he would probably run into issues, because changing the database schema, customizing the system models with a help of inheritance after the initial migrations had been run always leads to some troubles with cross dependencies, integrity errors, etc.

2. InitialSeverProfile
   This model represents the initial data that the user needs to enter. According to the InitialServerProfile, the application will connect to the remote instance, set up the environment, add users and create the Server model. After this, the Initial one could be deleted (later on this process will be automated).

3. Server
   This one is the core part of the system. It stores all the data related to the server instance. All the server's metrics are also stored here. It is linked to two other tables: Thresholds And URLs.

4. Threshold
   This one represents the thresholds for all the server's metrics. All of them are written in percentages.

5. URL
   This one is used to save the URLs that correspond to this website to check their SSL certificates users and check the HTTP status codes.

6. Session. The system model that is used by Django to manage user sessions.

7. Unknown session
   This one is used to store all the unknown systems. The corresponding script on the server-side creates the instance of this model if the system is not yet supported by our application.

8. Groups and Permissions. Those are used by the Django framework to better manage users and authorization flow.

9. BotUserLink
   This model is used to link current users in the system to the external chatbot user. Could be used with Messenger, Telegram, and other platforms.

10. LogEntry
    This one is responsible for storing the history of changes for all the objects in the system.

6.4 REST

REST states for the representational state transfer which is an architectural approach that describes how the systems can communicate through the web. The communication is done using textual representations of data objects in a stateless manner. This means that every request is contextually independent from another.

There are lots of benefits in stateless architecture, but the most important one is that such an approach is suitable to develop applications with a high load. The server doesn't need to store some client's session related information and cares about clearing it if the session is broken for some reason. Although it may require to include some additional data in each request.

The access to the data is done through a URL, so each resource that can be accessed from the client has its own URL. When the client retrieves the resource, the corresponding data is returned. It is not wrapped in some XML (like SOAP protocol does), but just the JSON object with needed fields is returned. It's relevant to mention that all the access to the data is done through the HTTP protocol. And client can manipulate data with the following HTTP methods:

- GET
- PUT
- POST
- DELETE

So, in this way REST fully supports CRUD, which is a notion that represents operations that are necessary to implement in each persistent storage application. CRUD stands for the following type of operations:

- CREATE

  This one allows a client to create a new resource on the server's data storage. In all SQL based databases this one is equivalent to INSERT.

- READ

  This one allows the client to read or to search the data by querying it or applying different filters.

- UPDATE

  This one allows the client to update some existing objects in the server's data storage.

- DELETE

  Obviously, the last one is responsible for deleting objects. So, the REST architecture is simple to implement and use. Meanwhile, it provides easy access to all the resources on the server.

6.4.1 Django Rest Framework

Django Rest Framework is widely used to provide the REST functionality using Django. It allows to quickly build the REST interface without code repeating, so following DRY principle. DRF is linked with Django itself, so it has built-in support for Django ORM. There are some main notions [*Django Rest Framework*]:

1. Serialization

   With help of so-called serializers, it is possible to validate the data that comes to or from our application. The serializer object receives the JSON (Python dictionary) object and runs the validation on provided fields. The main advantage of DRF serializers is that they support Model serialization. So, the data is automatically validated according to the rules that are described in the model.

2. Views

   Views are classes that implement the business logic of an application. But in some cases, especially when building some REST interface, the logic is pretty straightforward. The system just needs to give access to all the resources it has. And as I mentioned before, DRF supports the DRY principle, so there is no need to invent some resource views every time you create a REST interface.

   There are tons of built-in View classes from which the developer can inherit. They also support linking with a Model instance, so they can give access to the data in the database directly. Using inheritance the developer can control the behavior of views at the desired level of abstraction. The simplest implementation of CRUD:

   ```
   class SnippetList(generics.ListCreateAPIView,
                     generics.RetrieveUpdateDestroyAPIView):
       queryset = Snippet.objects.all()
       serializer_class = SnippetSerializer
   ```

   This will automatically create all the needed endpoints to support CRUD operations.

   Of course, when the developers need to customize some operations, they are allowed to do so by inheriting from other classes or redefining the existing implementations of provided default methods.

3. Authentication and authorization

   DRF also supports different ways to authenticate and authorize the user. There are many flows of auth, the most direct one is to specify the permission classes on the view level:

   ```
   permission_classes = [permissions.IsAuthenticatedOrReadOnly]
   ```

   If there is a need to customize those, a developer can create custom object-level permissions. They're widely used in systems, where it comes to the user management. In my application, that's not the case, but sometimes there is a need to have admins, content managers, writers, super admins, and others. They all have different permissions and that could be managed by creating the custom object-level permissions and then checking them in views.

4. Routers

   This part of DRF simplifies the routing of web applications. As it was mentioned before, the REST architecture is about giving access to the resources it has and each resource has its URL. It could be imagined what a mess it could be to define all the needed URLs for the large system and what would be the consequences of changing some of them.

Routers simplify all the processes by creating automatically all the URLs for the view. The example of defining a router and registering views:

```
router = DefaultRouter()
router.register(r'servers', views.ServerView)
router.register(r'users', views.UserView)
```

By registering the view with some prefix (servers and users in this case), all the needed URLs for supporting CRUD will be created (of course, if view supports all those operations).

6.4.2 Integration with frontend

As it was mentioned before, Django Admin is a powerful tool to manage the database, but it is too risky to allow users to interact with it. So, my application supports the integration with the front-end. By providing a REST interface to all the resources I am enabling the possibility to integrate almost all the modern web frameworks with my app. Using Angular, React, or Vue.js it is possible to build a full server management system. Moreover, the end-user can choose among different web templates the one they need, not to bother the system with useless charts and diagrams.

6.5 Celery and Celery beat

There are some long-running tasks like server enrollment that may take several minutes to be completed. It is not a good idea to force users to wait until this task is completed. But in the synchronous approach, it is the problem, because the process that processes the request needs to wait until the task is completed and the user has to wait, while the whole system from the user perspective gets stuck [*Celery - Distributed Task Queue*].

Here comes Celery – a simple, flexible, and reliable distributed system to process vast amounts of messages, while providing operations with the tools required to maintain such a system. It's a task queue with a focus on real-time processing, while also supporting task scheduling (will come to that later).

It has the following benefits [*Celery - Distributed Task Queue*]:

1. Simplicity

   Celery is a really simple tool to configure and use. The basic setup looks so:

   ```
   os.environ.setdefault("DJANGO_SETTINGS_MODULE",
                         "mon.settings")
   app = Celery("mon")
   app.config_from_object("django.conf:settings",
                          namespace="CELERY")
   app.autodiscover_tasks(lambda: settings.INSTALLED_APPS)
   ```

   And some environment variables in the settings (CELERY_BROKER_URL and CELERY_RESULT_BACKEND) with CELERY prefix, so it can find them.

2. Availability

   Celery is a highly available service because it supports automatic retrying when there are some connectivity issues or failures, as well as supporting replication.

3. Speed

   It is relatively fast, providing a possibility to run millions of tasks in a minute on a single process.

4. Flexibility

   Almost every part of Celery can be extended or used on its own, Custom pool implementations, serializers, compression schemes, logging, schedulers, consumers, producers, broker transports, and much more.

In my system Celery is used for the server enrollment task (not to bother users waiting until the completion) and checking the updating the server's resources metrics. The first task just runs asynchronously when the initial server has been created. But the second one needs to be triggered every minute (or the time the user specifies).

For scheduling celery tasks Celery Beat tool can be used. It is supported by Celery out of the box and it is amazingly flexible. It supports crontab way of setting up the scheduling:

```
app.conf.beat_schedule = {
    'add-every-monday-morning': {
        'task': 'tasks.add',
        'schedule': crontab(hour=7, minute=30, day_of_week=1),
        'args': (16, 16),
    },
}
```

This task will be executed every Monday morning at 7:30 a.m. Moreover, it supports solar scheduling.

The Celery and Celery Beat architecture is simple and looks so [Mariano, 2019]:

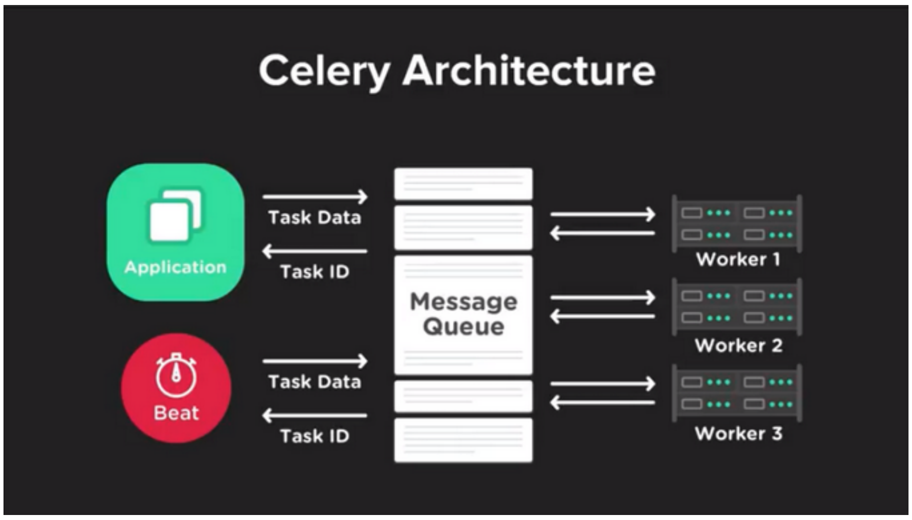


FIGURE 6.3: Celery Architecture

6.6 Docker and Docker-Compose

When dealing with the development process it is important to have similar environments on the server and locally, so all the tests and the system behavior, in general, are similar. When the application grows it could become a mess to create the environments manually for each instance.

Docker is a tool for managing the environments of the application. It provides the ability to build a so-called container and running the application isolated within it. The containers are lightweight and should have all the libraries and dependencies to run the application. This way it doesn't depend on what host system it is running.

Docker has a client-server architectural approach, so the client talks to the Docker daemon. They could run on one system or the client could be connected to the remote daemon. [*Docker Engine*]

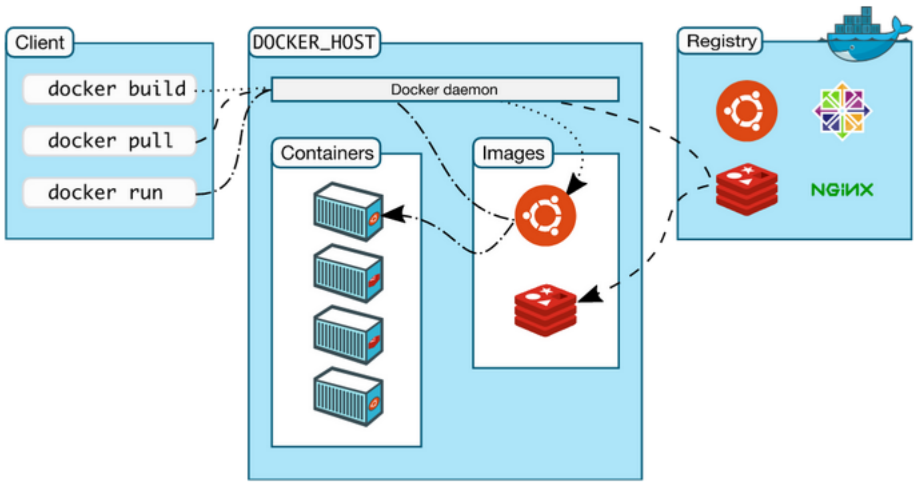


FIGURE 6.4: Docker Compose Architecture

There are some main parts in it:

1. The docker daemon listens for all the commands from the client and manages the docker objects (such as containers, images, etc.)

2. The docker client is the service that accepts all the end-user commands (i.e. docker run, docker build, etc.) and sends them to the docker daemon.

3. Docker objects

   There are some of them – like containers, images, networks, volumes, and plugins:

   - Image is a core part and it is basically a read-only instruction about how the system should be built. Often, the images are created based on others (for example the Ubuntu or CentOS images can be used) and then some customization is made.
   - Containers are just runnable instances of images. They could be created, run, deleted, stopped, or moved using the Docker client (either Docker API or CLI).

4. Docker registry is the place where all the images could be stored and managed.

The docker image of this application is based on the python:3.7-alpine which is very lightweight. All the needed packages are installed later with the "apk add" command.

Docker is cool but there is usually a need to have several containers within one application. This is also the case here. Then the Docker-Compose tool can be used. It is basically a YAML-based configuration tool that allows the developer to maintain multiple docker containers with just one configuration file. [*Docker Compose*] In my application, there are several containers: database (based on mysql:5.7 image), core (the application container, based on the python:3.7-alpine), celery, and celery-beat (also based on the python:3.7-alpine), and redis (based on redis:5-alpine).

The best part of it is that Compose works in all environments: production, staging, development, testing, as well as CI/CD workflows.

Chapter 7

Summary

In conclusion, it could be said that my application is a great combination of technologies that aim to improve the experience of maintaining servers by giving an easy solution for monitoring their resources.

The main idea of a project was to provide a back-end application that would give a user a flexible set of functions with the potential of different front-end frameworks integration, scaling possibilities, and out-of-the-box use. It was fulfilled with the help of the Django framework as the core part of the solution, Celery queue as the asynchronous supporter, and Docker + Docker-Compose as the service for setting up the environment.

There is a potential for improvements and it consists of the following steps:

- Add the unit testing with full coverage. It will allow maintaining the application easier when the codebase will grow.
- Choose a cloud provider and host the solution. This one is easy to implement because of the Docker-Compose tool which is responsible for setting up both local and server environments.
- Create a basic front-end. It is possible to create a default front-end using Django templates. It will replace the Django admin interface giving more control under the user's operations.

Nevertheless, the solution turned out to be working, and despite possible improvements, it already able to improve the experience of monitoring server resources.

Bibliography

Amazon CloudWatch. URL: https://aws.amazon.com/cloudwatch/.

Celery - Distributed Task Queue. URL: https://docs.celeryproject.org/en/stable/.

Cloud Monitoring. URL: https://cloud.google.com/monitoring/.

Django Models. URL: https://docs.djangoproject.com/en/3.2/topics/db/models/.

Django Rest Framework. URL: https://www.django-rest-framework.org/.

Docker Compose. URL: https://docs.docker.com/compose/.

Docker Engine. URL: https://docs.docker.com/engine/.

Gartner (2017). *Gartner Forecasts Worldwide Public Cloud End-User Spending to Grow 18% in 2021*. URL: https://www.gartner.com/en/newsroom/press-releases/2020-11-17-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-grow-18-percent-in-2021.

George, Nigel (2020). *Mastering Django*.

Holovaty, Adrian and Jacob K. Moss (2008). *The Definitive Guide to Django:Web Development Done Right*.

Lesonsky, Rieva (2017). *10 Ways to Get New Customers*. URL: https://www.sba.gov/blog/10-ways-get-new-customers.

Mariano, Antonio Di (2019). *How to run periodic tasks in Celery*. URL: https://antoniodimariano.medium.com/how-to-run-periodic-tasks-in-celery-28e1abf8b458/.

MVC Architecture - Model, View, Controller. URL: https://webshake.ru/oop-v-php-prodvinutyj-kurs/arhitektura-prilozheniya-i-pattern-mvc/.

Rud, Anna (2019). *Why and How Netflix, Amazon, and Uber Migrated to Microservices: Learn from Their Experience*. URL: https://www.hys-enterprise.com/blog/why-and-how-netflix-amazon-and-uber-migrated-to-microservices-learn-from-their-experience/.

What Is Load Balancing? (2018). URL: https://www.nginx.com/resources/glossary/load-balancing/.

Windows Server Administration Fundamentals (2011). John Wiley and Sons, Inc.

Wirtz, Bernd W. and Peter Daiser (2018). “Business Model Development: A Customer-Oriented Perspective”. In: *Journal of Business Models* 6.3, pp. 24–44.