Project Hand-over Manual

Electricity Monitoring

K(3-1)

Contents

Introduction

In this document we will explain our project in dept, and we will demonstrate with our user manual how our product needs to be installed and used.

Our team

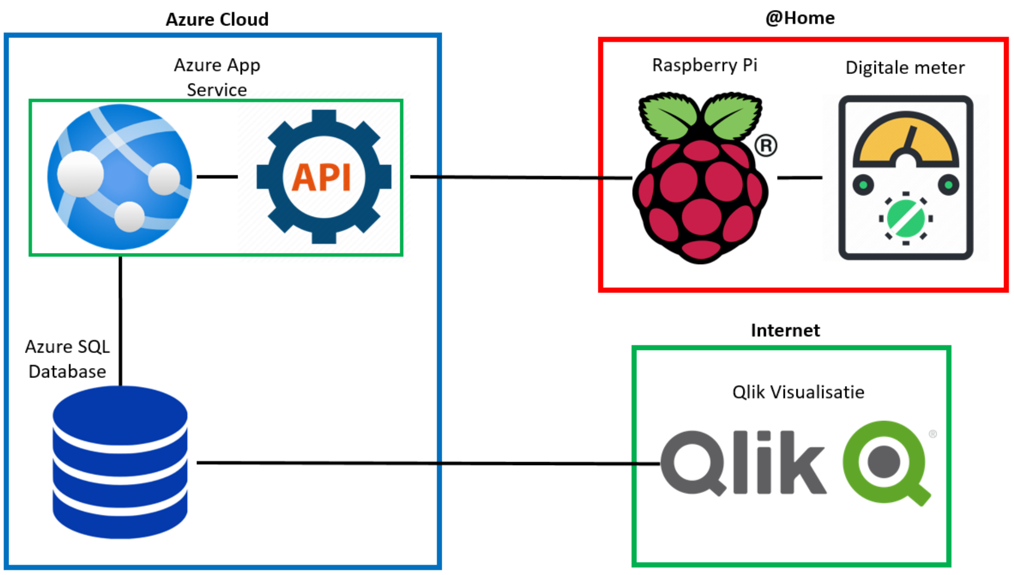
We are team K(3-1) or K2 and our team consists of 6 people. We have 3 APP students: Bert Van Staeyen, Marnix Brabants and Miel Goossens; 2 AI students: Ferre Van Hoof and Bram Henderickx; and 1 CCS Student: Wim Hembrechts. The fact that we are multidisciplinary makes us a wonderful team.

Description of our project

For our project we need to build a solution to read your Fluvius smart meter at home. This project is requested by the customer 3-IT from the Cronos-group. The result of our project will be an application where we can view our electricity usage to monitor everything because of the energy crisis we are in now.

You can find our solution here.

Overview of infrastructure



As you can see in the picture above, we've used the following components in our infrastructure:

- Azure App Service (Consisting of Docker image with Django app and built-in API)
- Azure SQL Database (MSSQL database)
- Qlik (Data visualisation)
- Raspberry Pi 4b (Data gathering and sending)
- Digital Smart Meter (Sagecom T211-D)

To create a web application, you obviously need a framework. We chose for Django as it is a high-level Python web framework. In this Django app we have embedded an API to make sure we can send data to our database in a save manner.

We've used Azure App Service to host this Django application, the app will be delivered by a docker container, which we've made with a custom Dockerfile. This docker image is then deployed to the Azure App Service using Terraform and GitLab-CI.

As database we chose an Azure SQL database to store all the data used by our application. This database is automatically deployed via the same GitLab pipeline used to create the docker image and deploy the Azure App Service.

To visualize all the data gathered by the Raspberry Pi from our smart meter we've used Qlik. We created 3 Qlik Dashboards that are embedded in our application. Users can see graphs created from data that is of the last 24hours, last 7days or last 30days.

The data we used was gathered by a Raspberry Pi 4b that is connected to our digital smart meter via a serial cable (RJ11 to USB). Every 15min we take and send 1 datapoint to our database via the API.

Manual

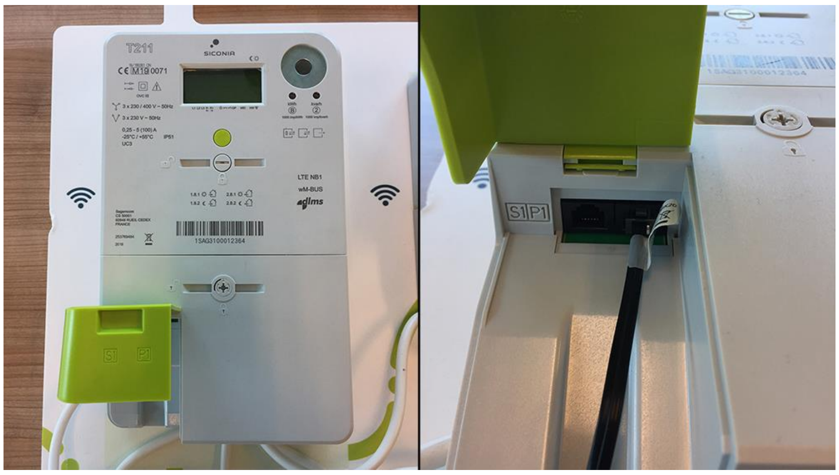
IOT

Hardware

Obviously we need some hardware to be able to read data from our Digital Smart Meter. A Raspberry Pi 4b alone is not enough. We'll need a cable to connect both devices. We're using a RJ11 to USB-A cable. The USB-A connector is plugged into our Raspberry Pi and the RJ11 connector is plugged into the P1 port on our smart meter.

The P1 port sends 1 datapoint every second. That data received is already formatted a bit formatted. Per line there is a code (OBIS-Code) and the value belonging to this code.

On our smart meter there is also another port namely the S1 port. This port continuously sends raw unformatted data. This port as much harder to use and get useful data from that's why we're not using this port.



Software

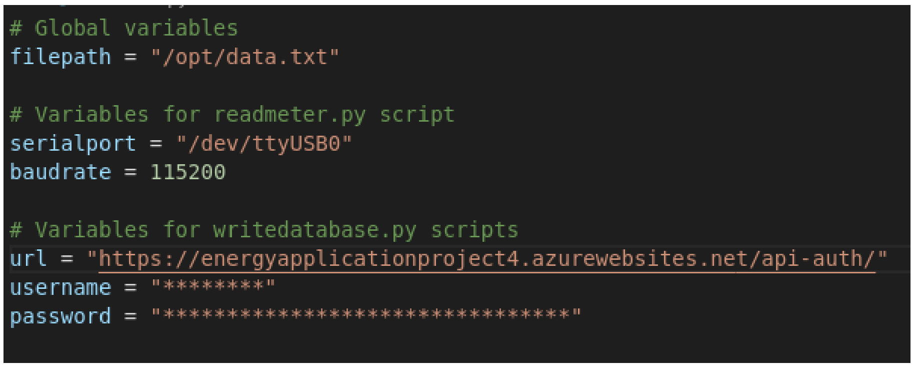
As mentioned we'll be using a Raspberry Pi 4b. We'll be using de default Raspberry Pi OS Lite 64-bit version on our Raspberry Pi. This operating system is based upon Debian 11 Bullseye. You can use Raspberry Pi Images to install this OS on the SD card.

The default OS is not enough. We of course need to install a few extra packages:

```
### Install required python modules
sudo apt install python3 python3-serial python3-pip
python3 -m pip install requests
```

Variables

The scripts that will be running on our Raspberry Pi needs some variables. Of course it is possible to declare these variables inside the scripts itself. But if we place these variables in a separate file, we have all variables in 1 place and we don't need to declare variables multiple times if they are used more than once. This file is than imported by both scripts.



```python
# Global variables
filepath = "/opt/data.txt"

# Variables for readmeter.py script
serialport = "/dev/ttyUSB0"
baudrate = 115200

# Variables for writedatabase.py scripts
url = "https://energyapplicationproject4.azurewebsites.net/api-auth/"
username = "********"
password = "********************************"
```

Services

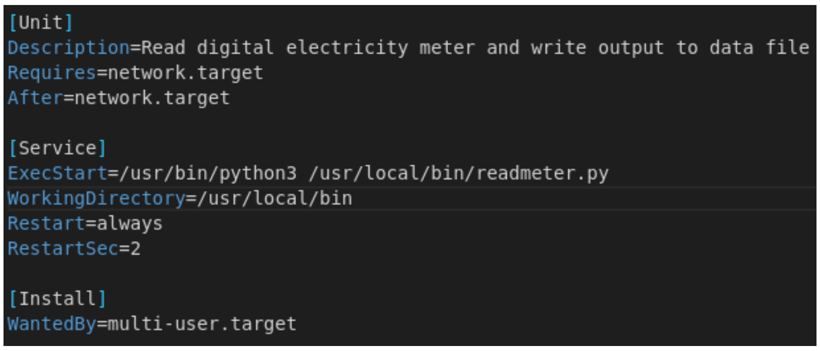
On the Raspberry Pi there will be running 2 python scripts at any time, 1 script that reads the data from the Digital Smart Meter and writes this data to a text file. The other script that reads this file and sends the data line by line to the database via an API.

The file where the data is temporarily written to acts as a buffer. If the network connection between the Raspberry Pi and the API is interrupted the data is still written to the file line by line. When the connection is restored, the data is read from this buffer file, transmitted via API and then deleted from this file. This way we do not have to worry about connection errors.

These 2 scripts are controlled by 2 services. These services only have 1 job, start the python script, and keep it running. If the script stops restart it.

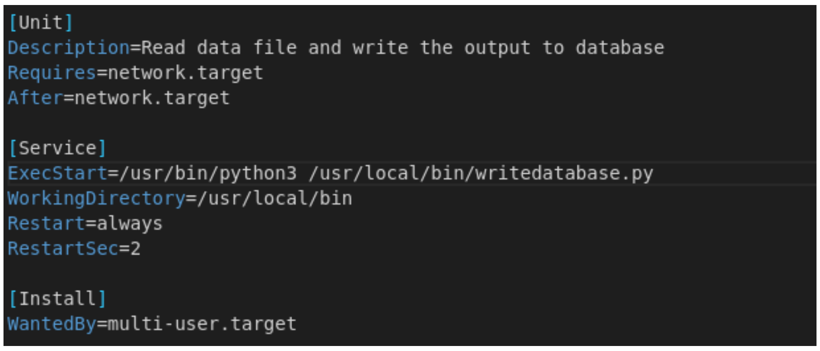
These services are located inside the directory: /etc/systemd/system/

Read meter service



```ini
[Unit]
Description=Read digital electricity meter and write output to data file
Requires=network.target
After=network.target

[Service]
ExecStart=/usr/bin/python3 /usr/local/bin/readmeter.py
WorkingDirectory=/usr/local/bin
Restart=always
RestartSec=2

[Install]
WantedBy=multi-user.target
```

Write to database service

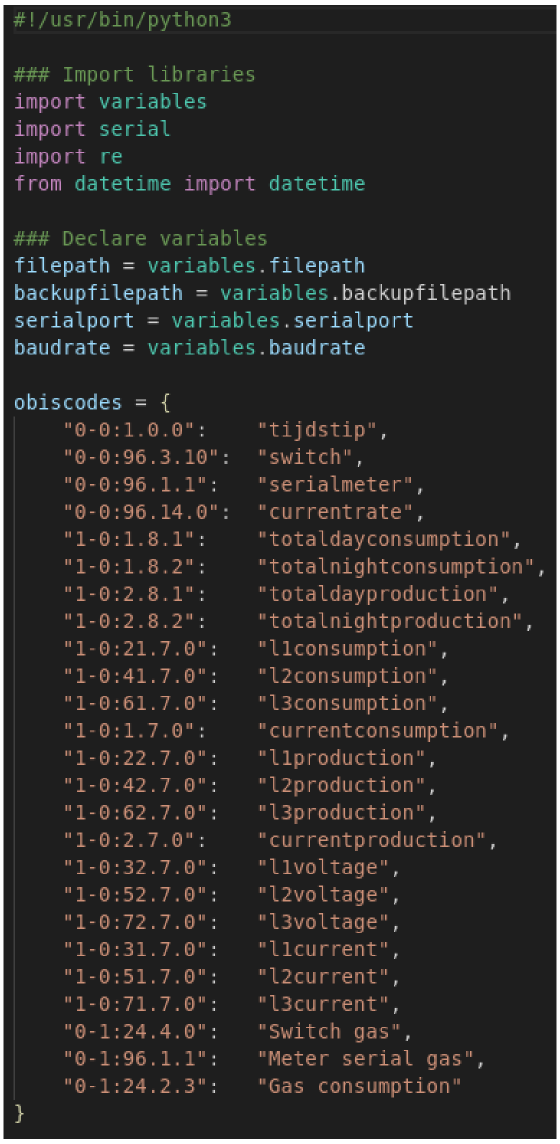


```
[Unit]
Description=Read data file and write the output to database
Requires=network.target
After=network.target

[Service]
ExecStart=/usr/bin/python3 /usr/local/bin/writedatabase.py
WorkingDirectory=/usr/local/bin
Restart=always
RestartSec=2

[Install]
WantedBy=multi-user.target
```

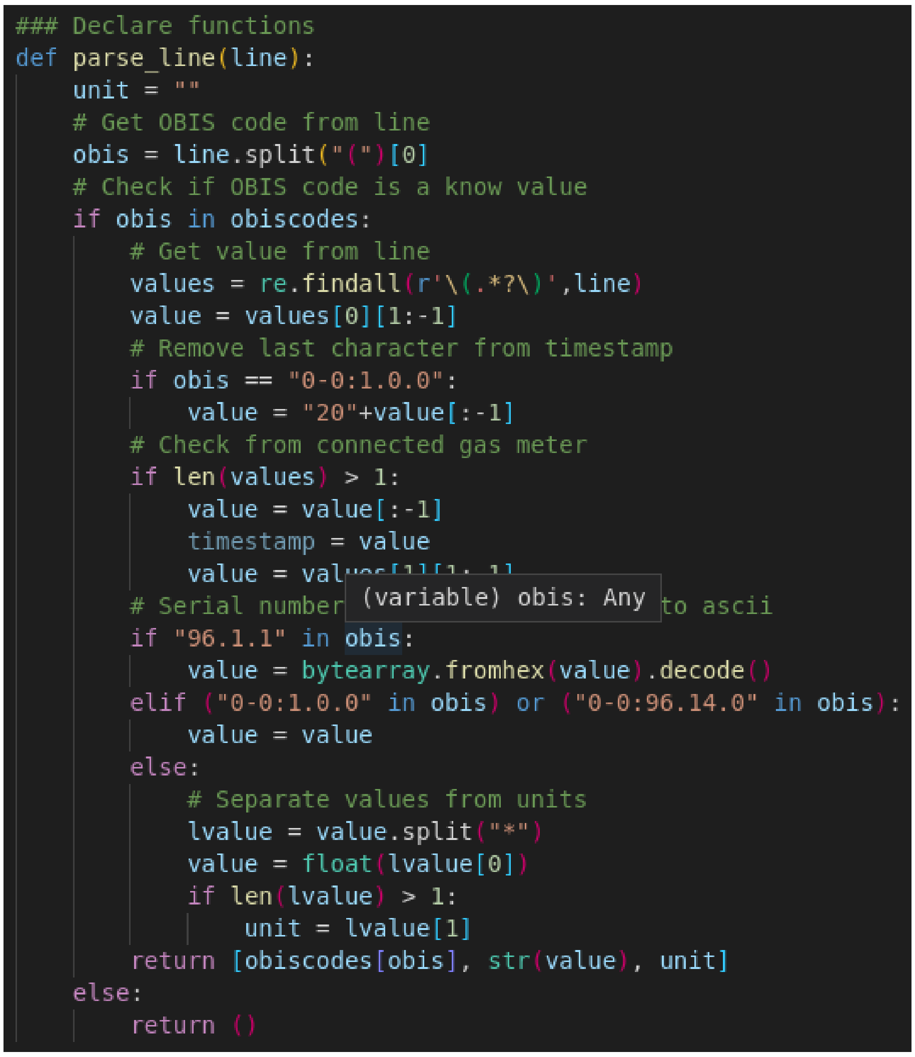
Scripts

Read meter script

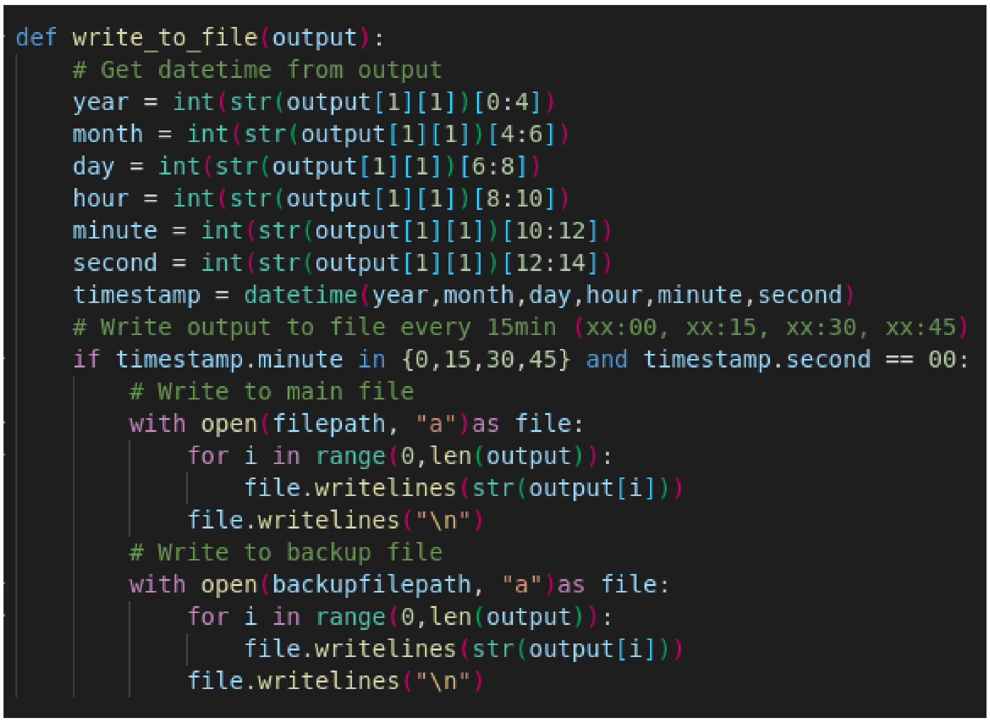
The following script reads the data from the Digital Smart Meter. Each datapoint contains a timestamp. If this timestamp is of the format xxh00m00s, xxh15m00s, xxh30m00s or xxh45m00s, so every 15min we take 1 datapoint from the meter. This datapoint is then formatted to make it easier to understand. After the formatting, this datapoint is written to the buffer file.



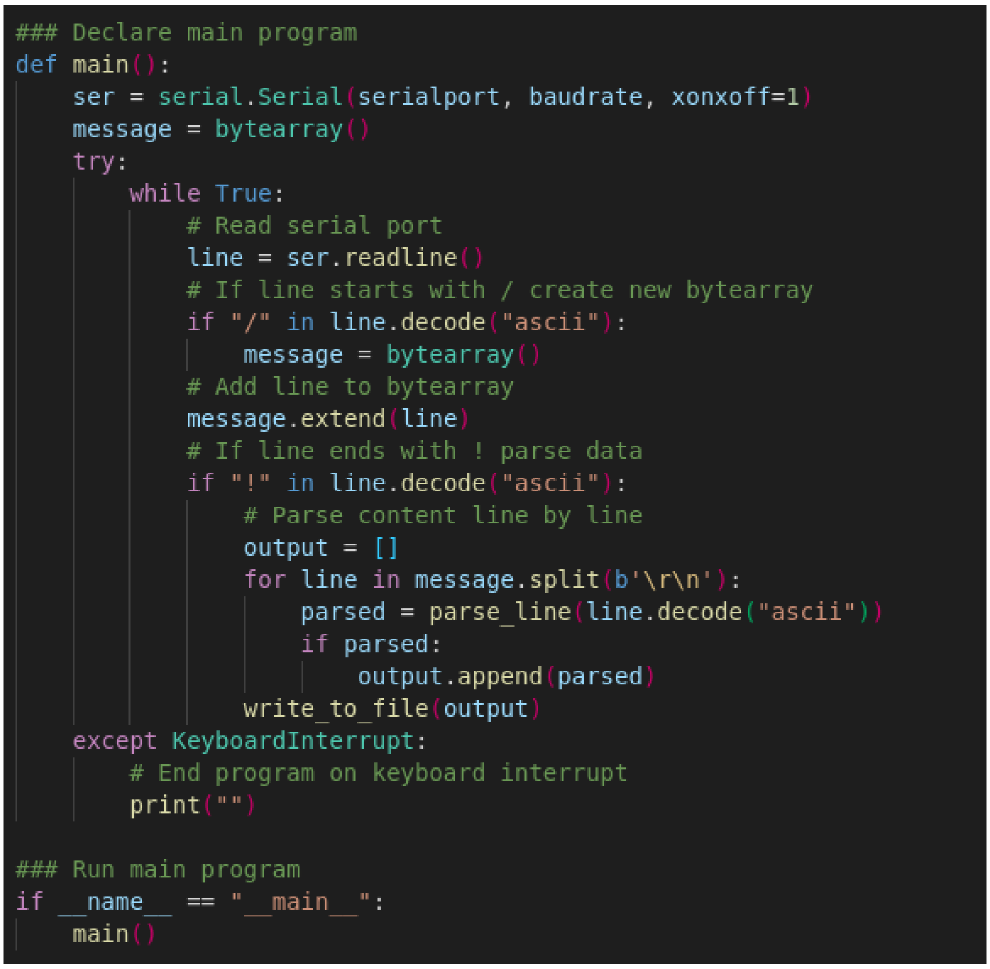
```python
#!/usr/bin/python3

### Import libraries
import variables
import serial
import re
from datetime import datetime

### Declare variables
filepath = variables.filepath
backupfilepath = variables.backupfilepath
serialport = variables.serialport
baudrate = variables.baudrate

obiscodes = {
    "0-0:1.0.0":    "tijdstip",
    "0-0:96.3.10":  "switch",
    "0-0:96.1.1":   "serialmeter",
    "0-0:96.14.0":  "currentrate",
    "1-0:1.8.1":    "totaldayconsumption",
    "1-0:1.8.2":    "totalnightconsumption",
    "1-0:2.8.1":    "totaldayproduction",
    "1-0:2.8.2":    "totalnightproduction",
    "1-0:21.7.0":   "l1consumption",
    "1-0:41.7.0":   "l2consumption",
    "1-0:61.7.0":   "l3consumption",
    "1-0:1.7.0":    "currentconsumption",
    "1-0:22.7.0":   "l1production",
    "1-0:42.7.0":   "l2production",
    "1-0:62.7.0":   "l3production",
    "1-0:2.7.0":    "currentproduction",
    "1-0:32.7.0":   "l1voltage",
    "1-0:52.7.0":   "l2voltage",
    "1-0:72.7.0":   "l3voltage",
    "1-0:31.7.0":   "l1current",
    "1-0:51.7.0":   "l2current",
    "1-0:71.7.0":   "l3current",
    "0-1:24.4.0":   "Switch gas",
    "0-1:96.1.1":   "Meter serial gas",
    "0-1:24.2.3":   "Gas consumption"
}
```



```python
### Declare functions
def parse_line(line):
    unit = ""
    # Get OBIS code from line
    obis = line.split("(")[0]
    # Check if OBIS code is a know value
    if obis in obiscodes:
        # Get value from line
        values = re.findall(r'\(.*?\)',line)
        value = values[0][1:-1]
        # Remove last character from timestamp
        if obis == "0-0:1.0.0":
            value = "20"+value[:-1]
        # Check from connected gas meter
        if len(values) > 1:
            value = value[:-1]
            timestamp = value
            value = values[1][1:-1]
        # Serial number [illegible] to ascii
        if "96.1.1" in obis:
            value = bytearray.fromhex(value).decode()
        elif ("0-0:1.0.0" in obis) or ("0-0:96.14.0" in obis):
            value = value
        else:
            # Separate values from units
            lvalue = value.split("*")
            value = float(lvalue[0])
            if len(lvalue) > 1:
                unit = lvalue[1]
        return [obiscodes[obis], str(value), unit]
    else:
        return ()
```



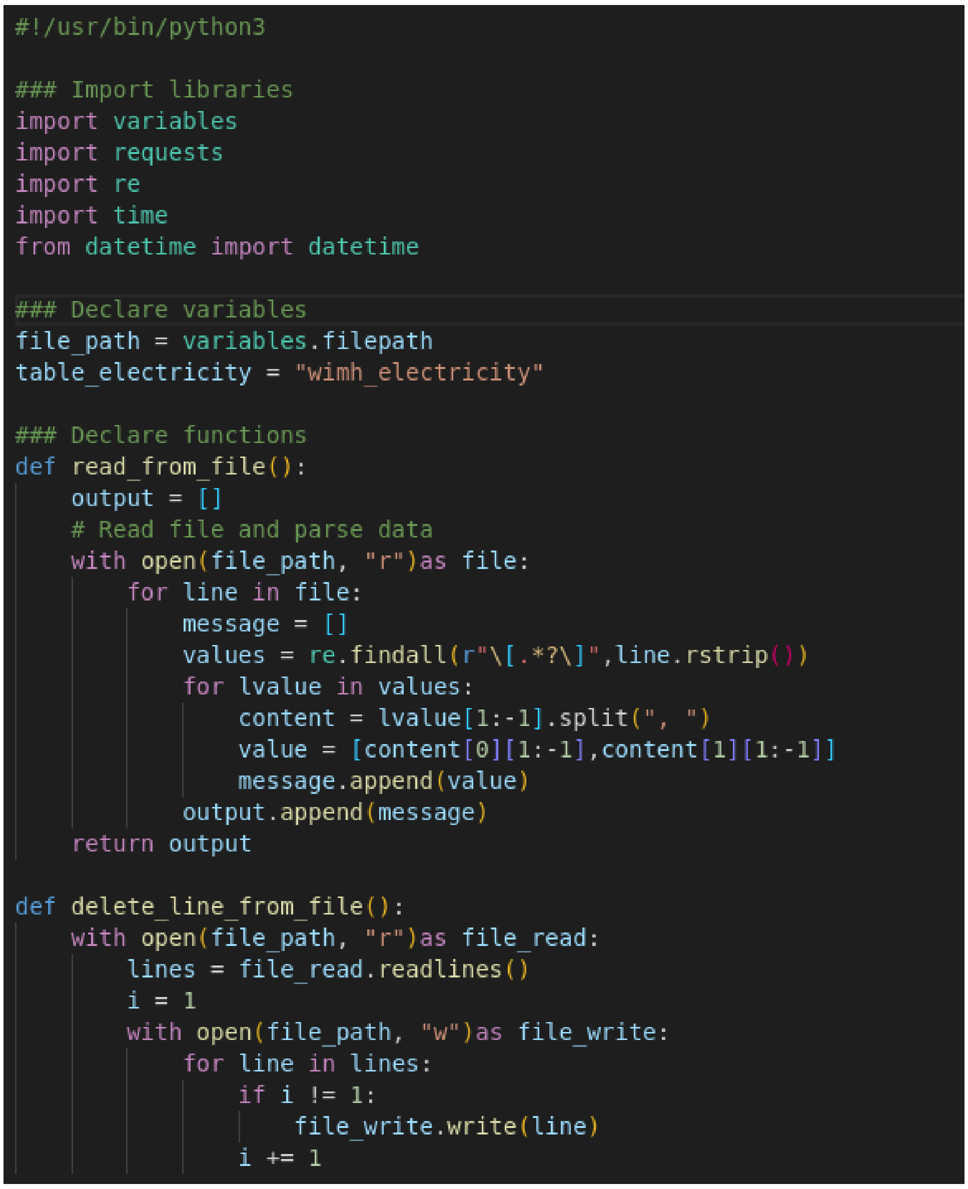
```python
def write_to_file(output):
    # Get datetime from output
    year = int(str(output[1][1])[0:4])
    month = int(str(output[1][1])[4:6])
    day = int(str(output[1][1])[6:8])
    hour = int(str(output[1][1])[8:10])
    minute = int(str(output[1][1])[10:12])
    second = int(str(output[1][1])[12:14])
    timestamp = datetime(year,month,day,hour,minute,second)
    # Write output to file every 15min (xx:00, xx:15, xx:30, xx:45)
    if timestamp.minute in {0,15,30,45} and timestamp.second == 00:
        # Write to main file
        with open(filepath, "a")as file:
            for i in range(0,len(output)):
                file.writelines(str(output[i]))
            file.writelines("\n")
        # Write to backup file
        with open(backupfilepath, "a")as file:
            for i in range(0,len(output)):
                file.writelines(str(output[i]))
            file.writelines("\n")
```



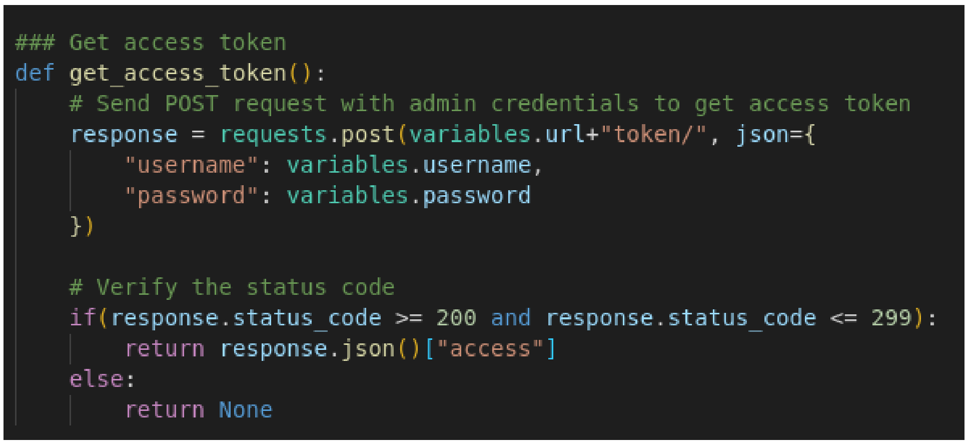
```python
### Declare main program
def main():
    ser = serial.Serial(serialport, baudrate, xonxoff=1)
    message = bytearray()
    try:
        while True:
            # Read serial port
            line = ser.readline()
            # If line starts with / create new bytearray
            if "/" in line.decode("ascii"):
                message = bytearray()
            # Add line to bytearray
            message.extend(line)
            # If line ends with ! parse data
            if "!" in line.decode("ascii"):
                # Parse content line by line
                output = []
                for line in message.split(b'\r\n'):
                    parsed = parse_line(line.decode("ascii"))
                    if parsed:
                        output.append(parsed)
                write_to_file(output)
    except KeyboardInterrupt:
        # End program on keyboard interrupt
        print("")

### Run main program
if __name__ == "__main__":
    main()
```

Write to database script

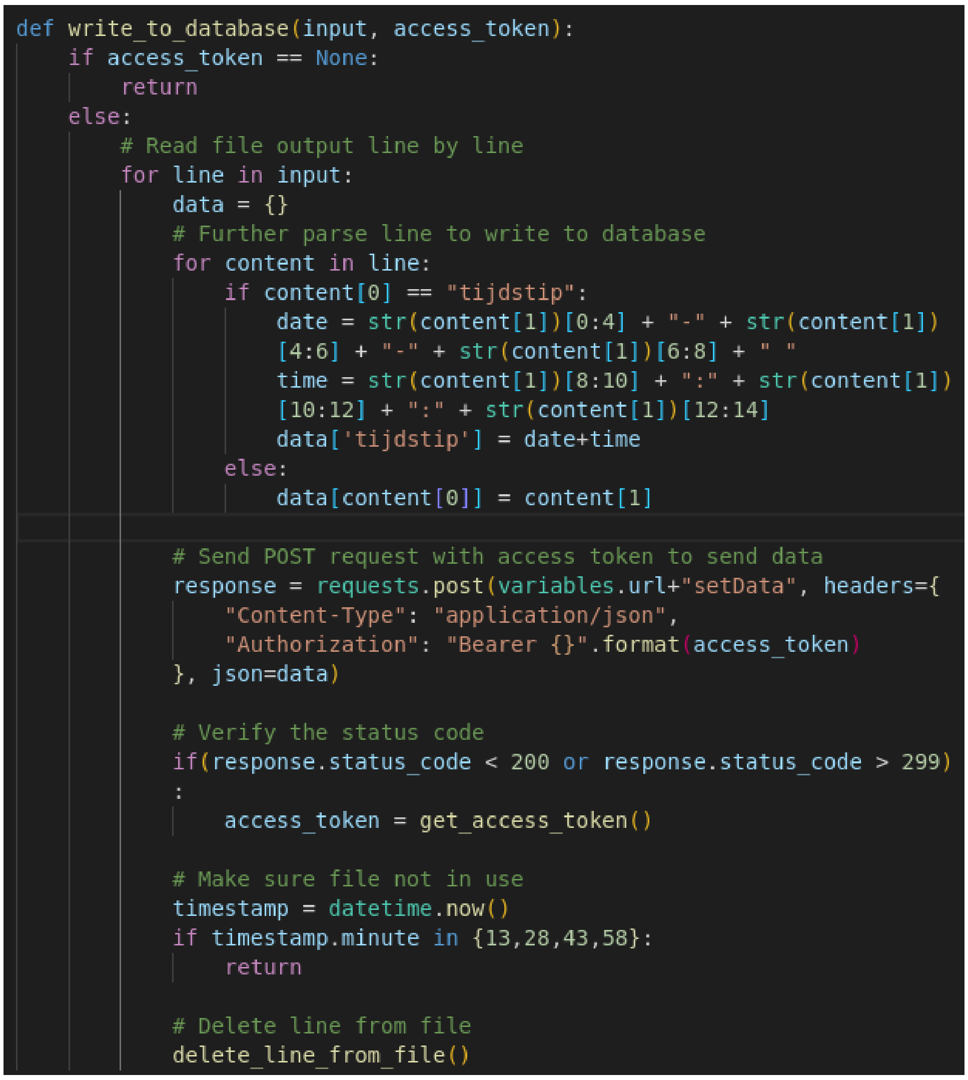
The next script takes care of the data transmission to the API. First, we make sure that the buffer file is not in use. If that is the case, the file is read to line by line. After we read the file, we request an API access token and afterwards we send the data in JSON format to the API. The API then writes the data to the database itself. After the data transmission is successful the line that was just written is delete from the buffer file. Then the cycle repeats for each line that is present in the file.



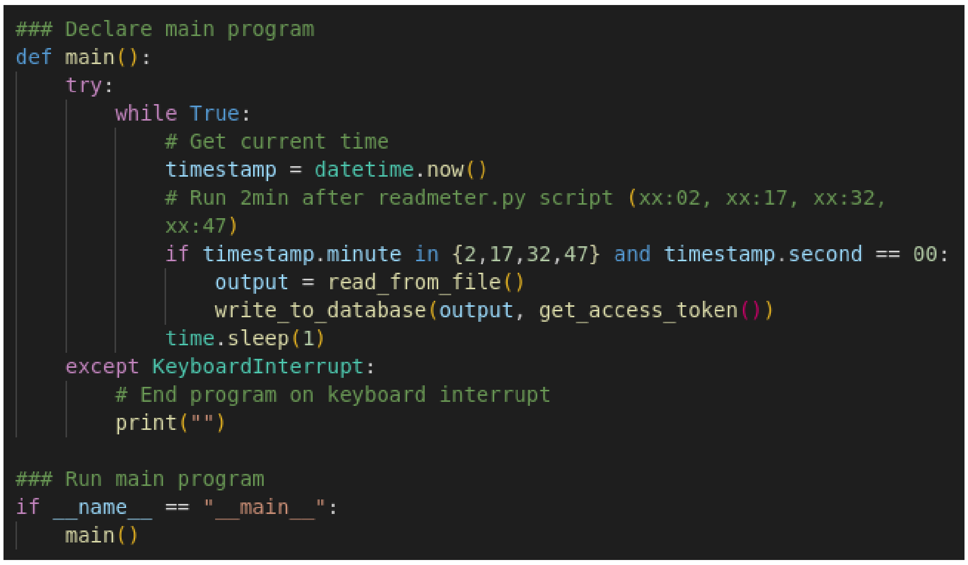
```python
#!/usr/bin/python3

### Import libraries
import variables
import requests
import re
import time
from datetime import datetime

### Declare variables
file_path = variables.filepath
table_electricity = "wimh_electricity"

### Declare functions
def read_from_file():
    output = []
    # Read file and parse data
    with open(file_path, "r")as file:
        for line in file:
            message = []
            values = re.findall(r"\[.*?\]",line.rstrip())
            for lvalue in values:
                content = lvalue[1:-1].split(", ")
                value = [content[0][1:-1],content[1][1:-1]]
                message.append(value)
            output.append(message)
    return output

def delete_line_from_file():
    with open(file_path, "r")as file_read:
        lines = file_read.readlines()
        i = 1
        with open(file_path, "w")as file_write:
            for line in lines:
                if i != 1:
                    file_write.write(line)
                i += 1
```



```python
### Get access token
def get_access_token():
    # Send POST request with admin credentials to get access token
    response = requests.post(variables.url+"token/", json={
        "username": variables.username,
        "password": variables.password
    })

    # Verify the status code
    if(response.status_code >= 200 and response.status_code <= 299):
        return response.json()["access"]
    else:
        return None
```



```python
def write_to_database(input, access_token):
    if access_token == None:
        return
    else:
        # Read file output line by line
        for line in input:
            data = {}
            # Further parse line to write to database
            for content in line:
                if content[0] == "tijdstip":
                    date = str(content[1])[0:4] + "-" + str(content[1])
                    [4:6] + "-" + str(content[1])[6:8] + " "
                    time = str(content[1])[8:10] + ":" + str(content[1])
                    [10:12] + ":" + str(content[1])[12:14]
                    data['tijdstip'] = date+time
                else:
                    data[content[0]] = content[1]

            # Send POST request with access token to send data
            response = requests.post(variables.url+"setData", headers={
                "Content-Type": "application/json",
                "Authorization": "Bearer {}".format(access_token)
            }, json=data)

            # Verify the status code
            if(response.status_code < 200 or response.status_code > 299)
            :
                access_token = get_access_token()

            # Make sure file not in use
            timestamp = datetime.now()
            if timestamp.minute in {13,28,43,58}:
                return

            # Delete line from file
            delete_line_from_file()
```



```python
### Declare main program
def main():
    try:
        while True:
            # Get current time
            timestamp = datetime.now()
            # Run 2min after readmeter.py script (xx:02, xx:17, xx:32,
            xx:47)
            if timestamp.minute in {2,17,32,47} and timestamp.second == 00:
                output = read_from_file()
                write_to_database(output, get_access_token())
            time.sleep(1)
    except KeyboardInterrupt:
        # End program on keyboard interrupt
        print("")

### Run main program
if __name__ == "__main__":
    main()
```

Application

Browser support: Chrome and Microsoft Edge work perfectly out of the box. Firefox and Brave however is possible but you do need to go to settings of the browser and turn off cookies. This is because we use a business license. This license can give some trouble when using Firefox or Brave. This issue should be resolved when using an enterprise license.

Stack

For our application we used python with a Django framework.

Because the graphs to display are from Qlik we also needed jQuery to make sure the embedding is done right.

For styling we used Tailwind CSS and Flowbite.

Django modules

In this chapter we will go over a couple of important files.

Django: user_management

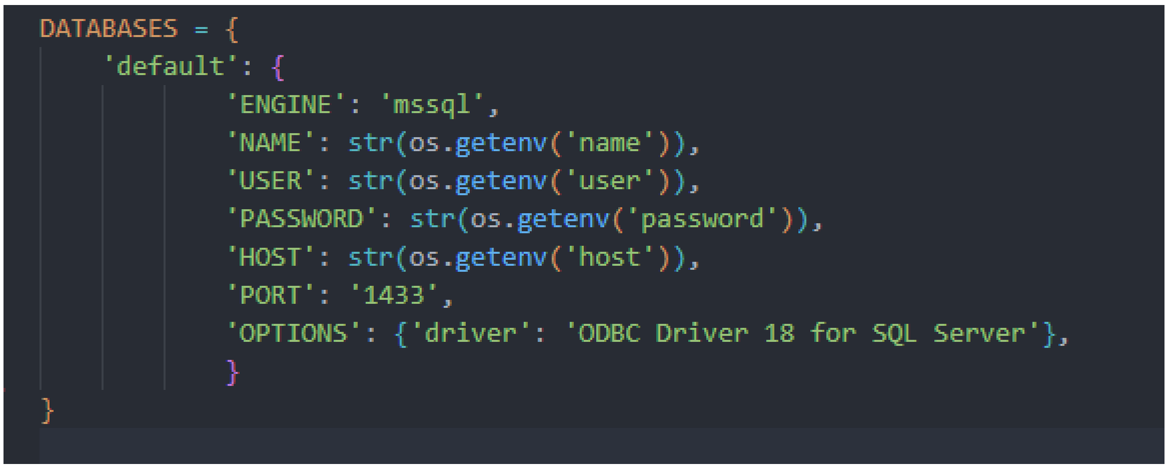
In Django you have a root module (App) which in our case is called: user_management.

This module takes care of backend related items.

Django: user_management – settings.py

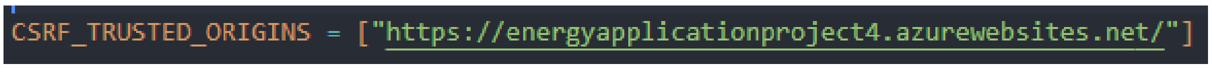
This folder contains the *settings.py*. Which is a file that is used for a lot of items:

- Database connection:



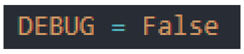
```
DATABASES = {
    'default': {
            'ENGINE': 'mssql',
            'NAME': str(os.getenv('name')),
            'USER': str(os.getenv('user')),
            'PASSWORD': str(os.getenv('password')),
            'HOST': str(os.getenv('host')),
            'PORT': '1433',
            'OPTIONS': {'driver': 'ODBC Driver 18 for SQL Server'},
            }
}
```

- Trusted Origins: (only requests from this origin will be accepted, for testing you might need to add localhost)



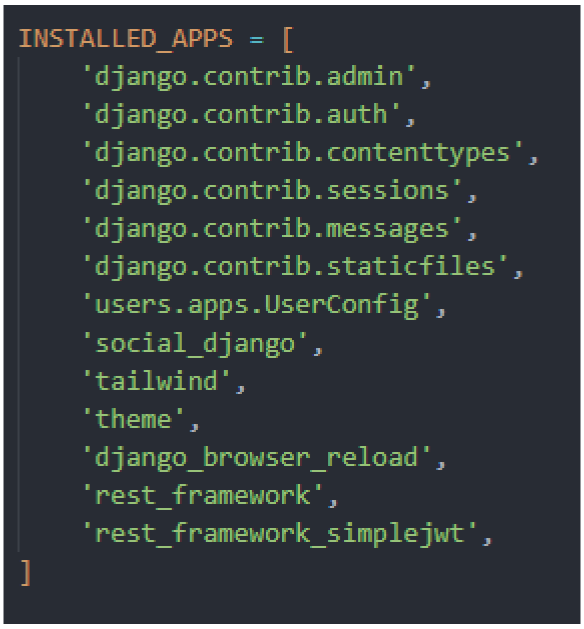
```
CSRF_TRUSTED_ORIGINS = ["https://energyapplicationproject4.azurewebsites.net/"]
```

- Debug: Debug true will give explanation on error.



```
DEBUG = False
```

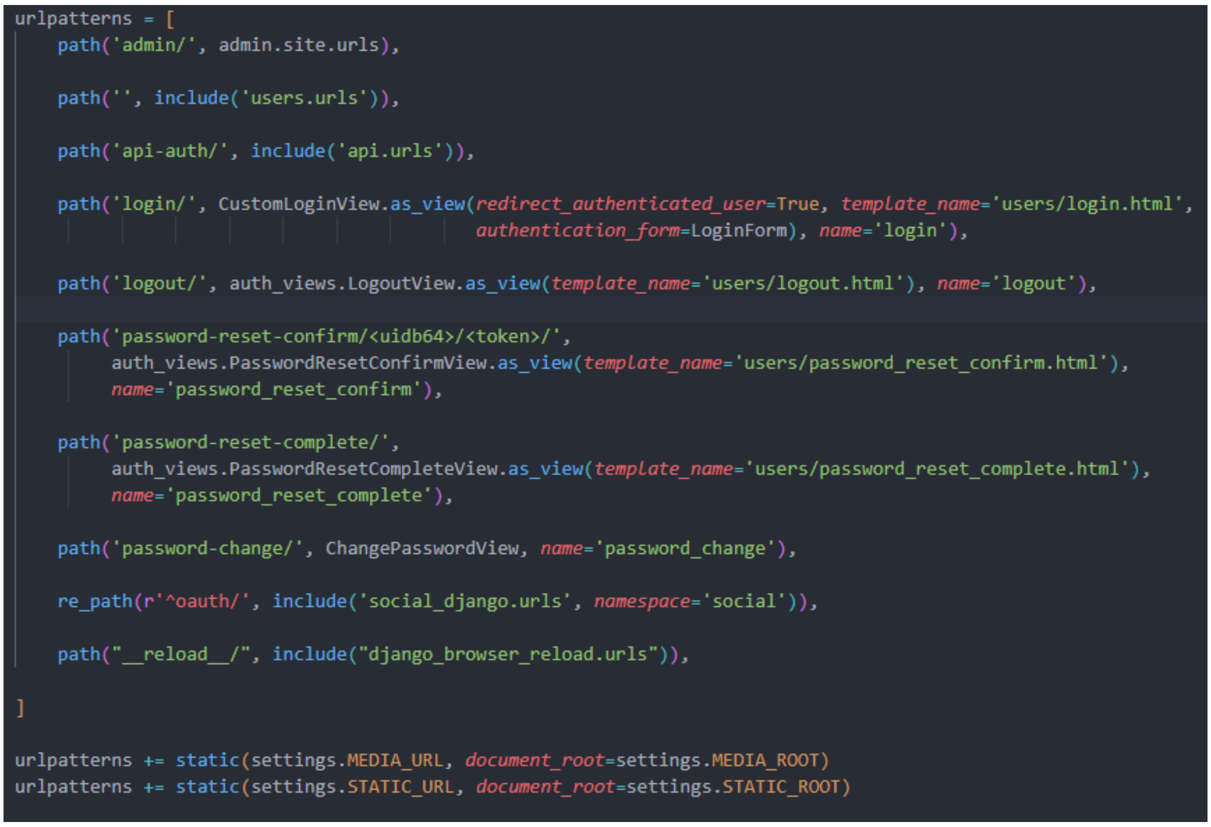
- Installed Apps: Packages needed for the application (does not equal requirements.txt)



```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'users.apps.UserConfig',
    'social_django',
    'tailwind',
    'theme',
    'django_browser_reload',
    'rest_framework',
    'rest_framework_simplejwt',
]
```

Django: user_management – requirements.txt
Requirements.txt is a file which contains all the dependencies.

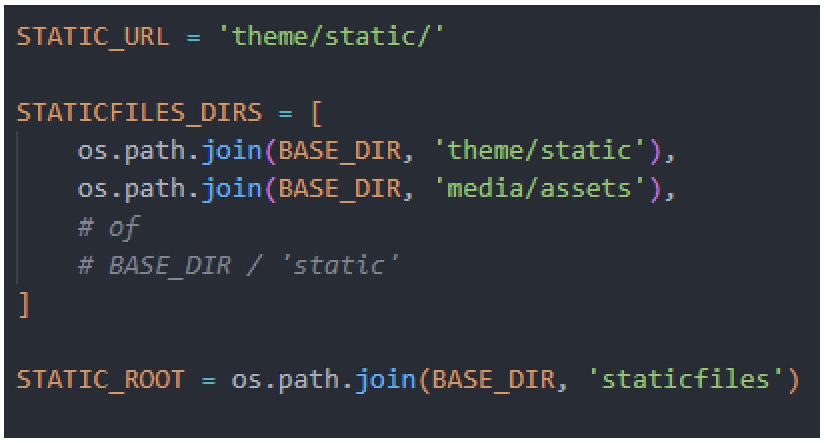
Django: user_management – urls.py
Within the user_management app you have a file named urls.py. This contains all the URLs that are available:



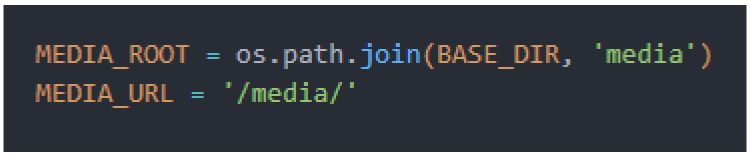
```python
urlpatterns = [
    path('admin/', admin.site.urls),

    path('', include('users.urls')),

    path('api-auth/', include('api.urls')),

    path('login/', CustomLoginView.as_view(redirect_authenticated_user=True, template_name='users/login.html',
                                            authentication_form=LoginForm), name='login'),

    path('logout/', auth_views.LogoutView.as_view(template_name='users/logout.html'), name='logout'),

    path('password-reset-confirm/<uidb64>/<token>/',
         auth_views.PasswordResetConfirmView.as_view(template_name='users/password_reset_confirm.html'),
         name='password_reset_confirm'),

    path('password-reset-complete/',
         auth_views.PasswordResetCompleteView.as_view(template_name='users/password_reset_complete.html'),
         name='password_reset_complete'),

    path('password-change/', ChangePasswordView, name='password_change'),

    re_path(r'^oauth/', include('social_django.urls', namespace='social')),

    path("__reload__/", include("django_browser_reload.urls")),

]

urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
urlpatterns += static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
```

Few important things to note:

- At the bottom we implement static and media, this contains styling and images. Extra settings for static and media can be found in the settings.py file near the bottom:



```python
STATIC_URL = 'theme/static/'

STATICFILES_DIRS = [
    os.path.join(BASE_DIR, 'theme/static'),
    os.path.join(BASE_DIR, 'media/assets'),
    # of
    # BASE_DIR / 'static'
]

STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')
```



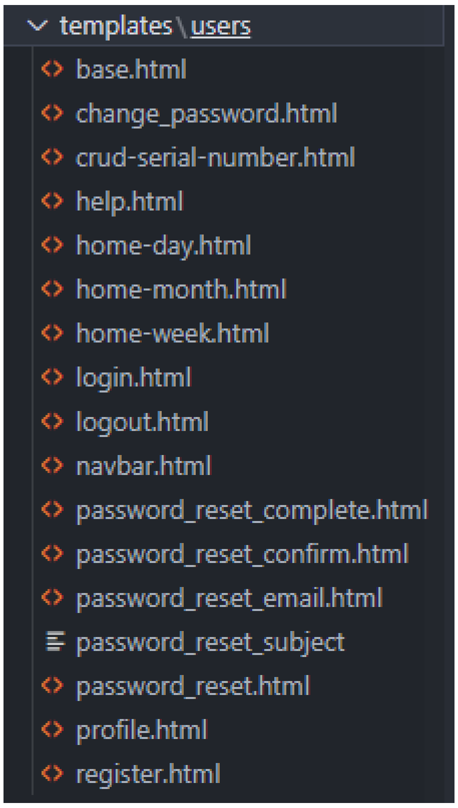
```python
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
MEDIA_URL = '/media/'
```

Django: users

As mentioned, user_management is the root of the application. But we also have another module named user. In this folder you will find everything related to the "frontend."

Django: users – templates folder

In this folder you will find html files used in the application.



```
templates\users
  base.html
  change_password.html
  crud-serial-number.html
  help.html
  home-day.html
  home-month.html
  home-week.html
  login.html
  logout.html
  navbar.html
  password_reset_complete.html
  password_reset_confirm.html
  password_reset_email.html
  password_reset_subject
  password_reset.html
  profile.html
  register.html
```

Django: users – forms.py

Every form that we use in the application is defined in this file.

This gives you the ability to validate a form fast.

Django: users – models.py

Models.py is used to create migrations. Django will look in this file when making new migrations. Every class will represent a table in the database. A lot of the classes that you will find are predefined classes by Django and usually describe something about the user.

We added a couple of extra ones:

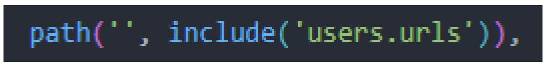
- Profile
    - This is a One-to-one relationship with a user. This way we can use the user in our own classes without breaking everything. It is a good habit to have this class if you want to link items to a user.
- SerialNumber
    - A serial number always belongs to a profile (owner). A serial number can have a name.
- WimElectricity (Last one in the file)
    - This class contains all the attributes which the Digital Meter will give to us.

Every other class is auto generated by Django and should not be modified.

Django: users – urls.py

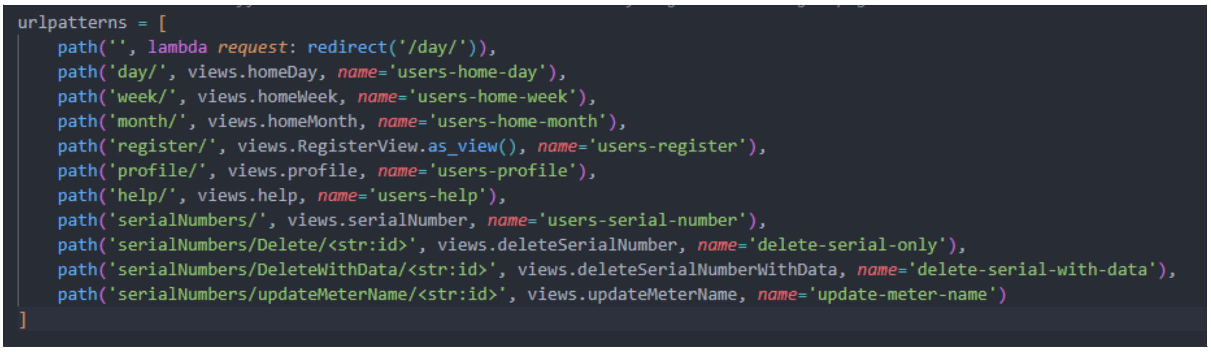
This file contains all the URLs.

In the *urls.py* within the user_management folder we can see the following code:



```
path('', include('users.urls')),
```

This means that every url in the user's module starts with a "/" and has no prefix. In the following code you can see all the URLs:



```
urlpatterns = [
    path('', lambda request: redirect('/day/')),
    path('day/', views.homeDay, name='users-home-day'),
    path('week/', views.homeWeek, name='users-home-week'),
    path('month/', views.homeMonth, name='users-home-month'),
    path('register/', views.RegisterView.as_view(), name='users-register'),
    path('profile/', views.profile, name='users-profile'),
    path('help/', views.help, name='users-help'),
    path('serialNumbers/', views.serialNumber, name='users-serial-number'),
    path('serialNumbers/Delete/<str:id>', views.deleteSerialNumber, name='delete-serial-only'),
    path('serialNumbers/DeleteWithData/<str:id>', views.deleteSerialNumberWithData, name='delete-serial-with-data'),
    path('serialNumbers/updateMeterName/<str:id>', views.updateMeterName, name='update-meter-name')
]
```

If you visit the root (example: "localhost:8000/") you will be redirected to /day/ (example: "localhost:8000/day/)

In the views file we will explain every url.

Django: users – views.py + templates

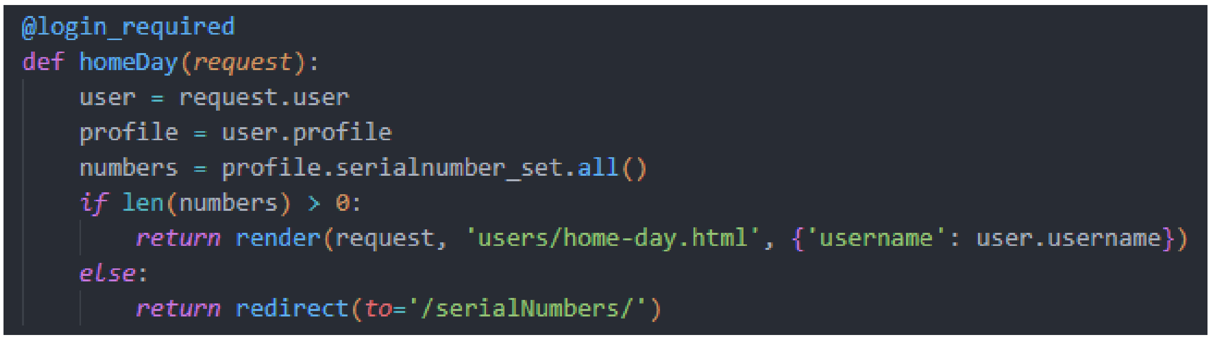
In this file all the logic happens.

Some functions in the views.py will have a "@login_required," if a user is not authenticated, they will be redirected to the login page before they can proceed to the given link.

homeDay + homeWeek + homeMonth

These three URLs are remarkably similar; therefore, we will discuss only *homeDay*, but this applies to homeWeek and homeMonth as well.

This will be triggered when visiting /day/ (/week/ for homeWeek and /month/ for homeMonth) and will trigger the following code in the *views.py*:



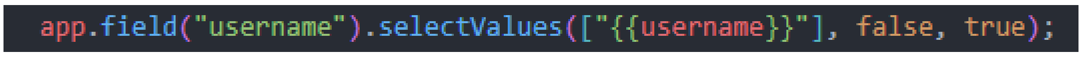
```
@login_required
def homeDay(request):
    user = request.user
    profile = user.profile
    numbers = profile.serialnumber_set.all()
    if len(numbers) > 0:
        return render(request, 'users/home-day.html', {'username': user.username})
    else:
        return redirect(to='/serialNumbers/')
```

In this code the user and his/her serial numbers are loaded and will be send to the template.

If a user has no serial numbers, he/she will be redirected to the serial numbers page where they first must add a serial number.

In the template you will find a script, this is where the embedding of Qlik happens.

The following line of code will give Qlik the user, they will then get the right data and create the graphs:



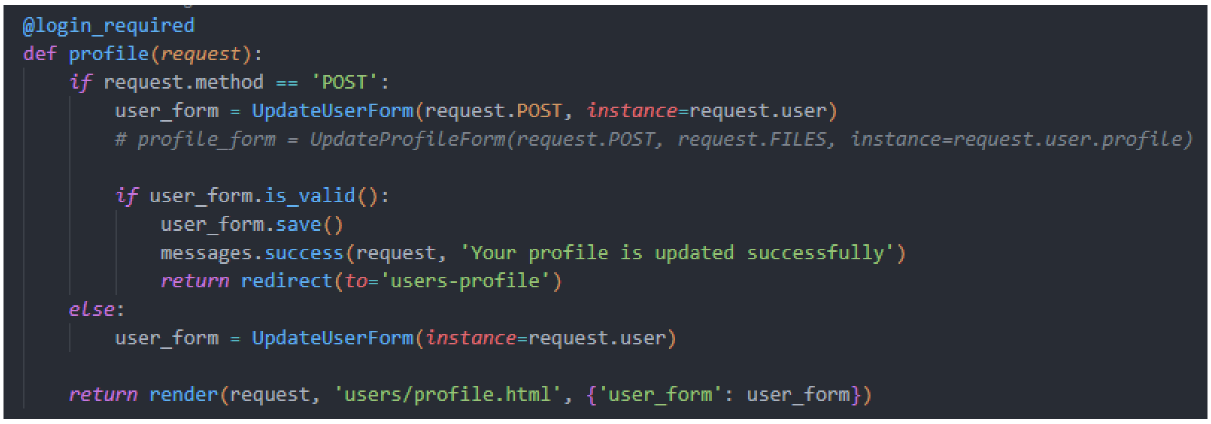
```
app.field("username").selectValues(["{{username}}"], false, true);
```

register

This will be triggered when visiting /register/

profile

This will be triggered when visiting /profile/ and will trigger the following code in the *views.py*:

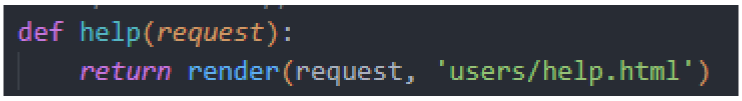


```
@login_required
def profile(request):
    if request.method == 'POST':
        user_form = UpdateUserForm(request.POST, instance=request.user)
        # profile_form = UpdateProfileForm(request.POST, request.FILES, instance=request.user.profile)

        if user_form.is_valid():
            user_form.save()
            messages.success(request, 'Your profile is updated successfully')
            return redirect(to='users-profile')
    else:
        user_form = UpdateUserForm(instance=request.user)

    return render(request, 'users/profile.html', {'user_form': user_form})
```

This will make a form with the data currently assigned to the user. The user can then modify and update his/her own profile.

help

This will be triggered when visiting /help/ and will trigger the following code in the *views.py*:

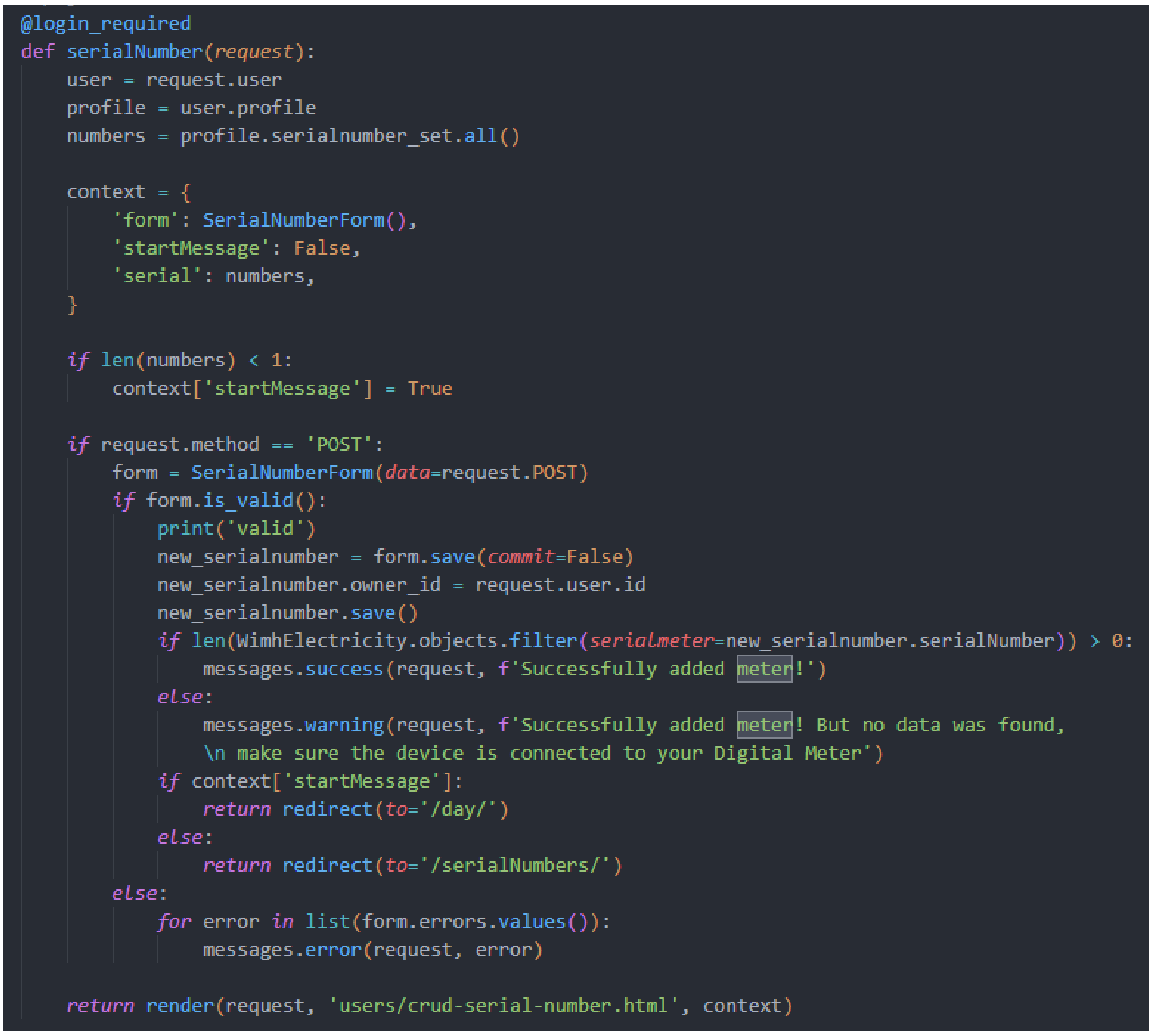


```
def help(request):
    return render(request, 'users/help.html')
```

This will show the user our help-page where they can find extra information. The user does not need to be authenticated to visit this url.

serialNumber

This will be triggered when visiting /serialNumber/ and will trigger the following code in the *views.py*:



```python
@login_required
def serialNumber(request):
    user = request.user
    profile = user.profile
    numbers = profile.serialnumber_set.all()

    context = {
        'form': SerialNumberForm(),
        'startMessage': False,
        'serial': numbers,
    }

    if len(numbers) < 1:
        context['startMessage'] = True

    if request.method == 'POST':
        form = SerialNumberForm(data=request.POST)
        if form.is_valid():
            print('valid')
            new_serialnumber = form.save(commit=False)
            new_serialnumber.owner_id = request.user.id
            new_serialnumber.save()
            if len(WimhElectricity.objects.filter(serialmeter=new_serialnumber.serialNumber)) > 0:
                messages.success(request, f'Successfully added meter!')
            else:
                messages.warning(request, f'Successfully added meter! But no data was found,
                \n make sure the device is connected to your Digital Meter')
            if context['startMessage']:
                return redirect(to='/day/')
            else:
                return redirect(to='/serialNumbers/')
        else:
            for error in list(form.errors.values()):
                messages.error(request, error)

    return render(request, 'users/crud-serial-number.html', context)
```

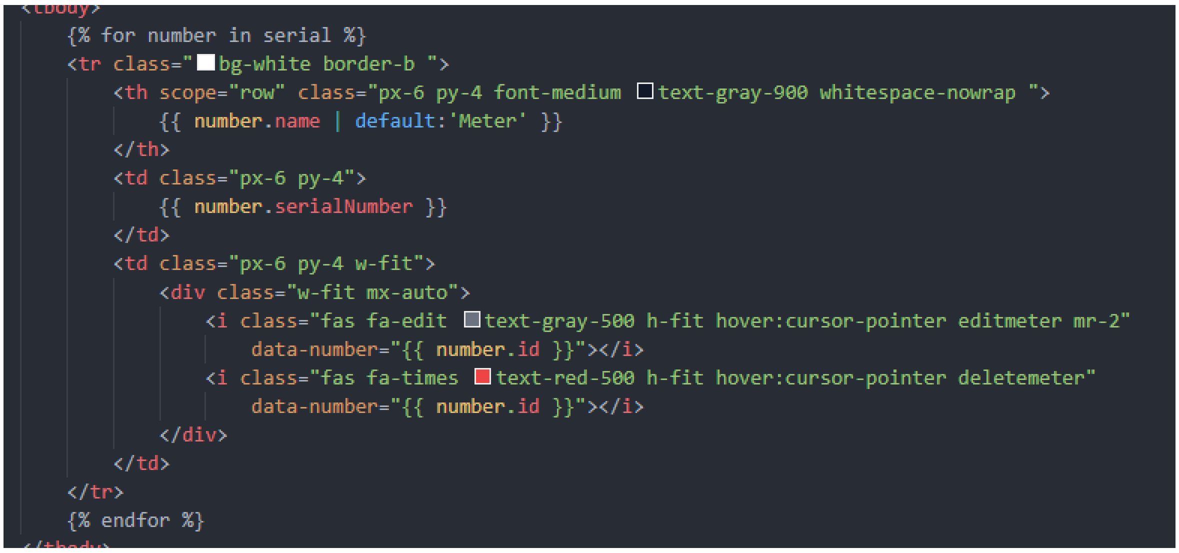
In this code we will create a form, this will allow the user to add a meter to their profile.

If the user has no meters, a start message will be shown to inform the user.

When the user adds a meter, a few things can happen:

- If the user added a meter but we do not have data of that meter. The user will be informed that the meter has been added successfully but that no data was found.
- In case the user added a meter, and we did find data, a success-message will be shown
- When the user adds its first meter, he/she will be redirected to the day page where they can start seeing their data.
- When a user already has a meter linked to its profile, he/she will stay on this page.

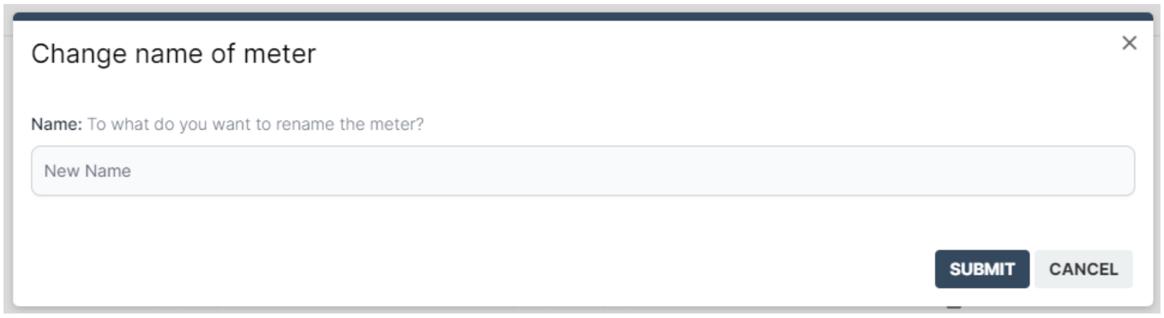
On the current template page, you can see 2 I-tags (icons):



```
<tbody>
    {% for number in serial %}
    <tr class="bg-white border-b ">
        <th scope="row" class="px-6 py-4 font-medium text-gray-900 whitespace-nowrap ">
            {{ number.name | default:'Meter' }}
        </th>
        <td class="px-6 py-4">
            {{ number.serialNumber }}
        </td>
        <td class="px-6 py-4 w-fit">
            <div class="w-fit mx-auto">
                <i class="fas fa-edit text-gray-500 h-fit hover:cursor-pointer editmeter mr-2"
                    data-number="{{ number.id }}"></i>
                <i class="fas fa-times text-red-500 h-fit hover:cursor-pointer deletemeter"
                    data-number="{{ number.id }}"></i>
            </div>
        </td>
    </tr>
    {% endfor %}
```

These tags will open modals:

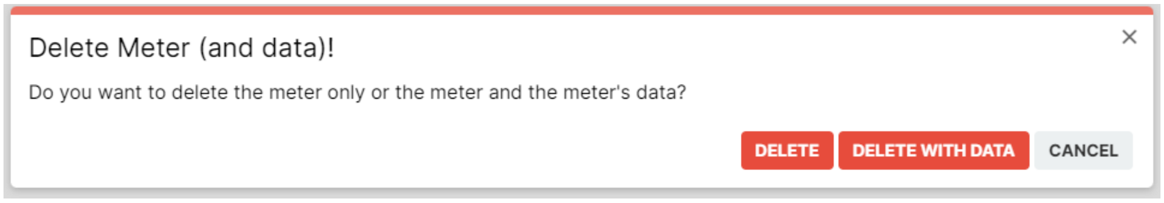
To change the name of the meter   To delete the meter



Change name of meter

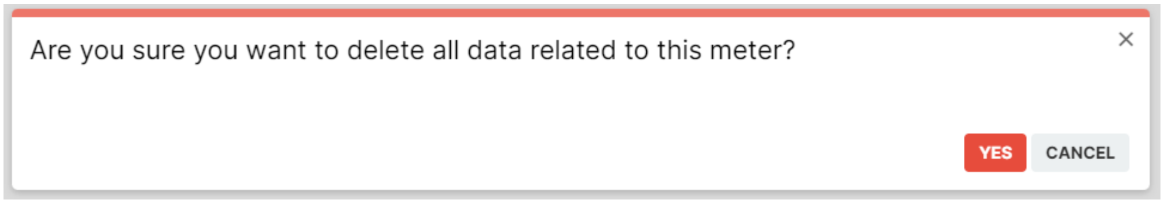
Name: To what do you want to rename the meter?

New Name



Delete Meter (and data)!

Do you want to delete the meter only or the meter and the meter's data?



Are you sure you want to delete all data related to this meter?

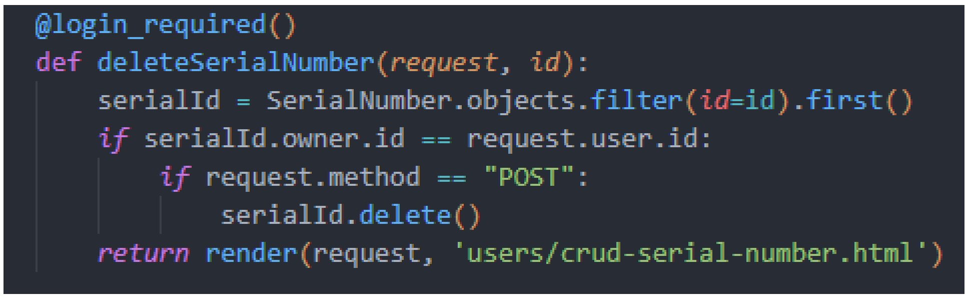
A user can delete or edit their meter.

When a user wants to delete their meter, they need to decide. They can choose to delete their meter, but keep the data connected to that meter. Or they can choose to delete the meter **with all the data** connected to that meter.

When they choose to delete their meter with data, they must confirm this once more since this action cannot be reversed.

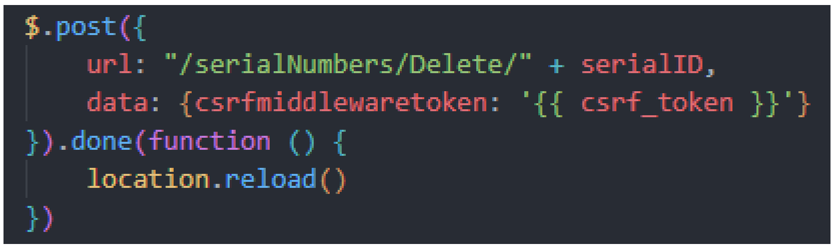
deleteSerialNumber

This will be triggered when visiting /serialNumber/Delete/(id) and will trigger the following code in the *views.py*:



```python
@login_required()
def deleteSerialNumber(request, id):
    serialId = SerialNumber.objects.filter(id=id).first()
    if serialId.owner.id == request.user.id:
        if request.method == "POST":
            serialId.delete()
    return render(request, 'users/crud-serial-number.html')
```

This code will only have an impact if the request method is POST, we use jQuery in our template to make this call:

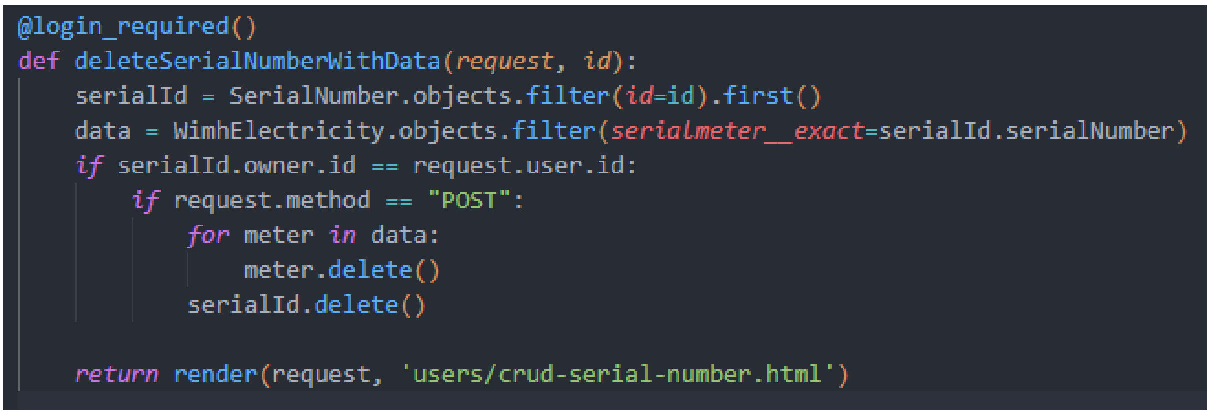


```javascript
$.post({
    url: "/serialNumbers/Delete/" + serialID,
    data: {csrfmiddlewaretoken: '{{ csrf_token }}'}
}).done(function () {
    location.reload()
})
```

This will only delete the meter but will keep the data.

deleteSerialNumberWithData

This will be triggered when visiting /serialNumber/DeleteWithData/(id) and will trigger the following code in the *views.py*:



```python
@login_required()
def deleteSerialNumberWithData(request, id):
    serialId = SerialNumber.objects.filter(id=id).first()
    data = WimhElectricity.objects.filter(serialmeter__exact=serialId.serialNumber)
    if serialId.owner.id == request.user.id:
        if request.method == "POST":
            for meter in data:
                meter.delete()
            serialId.delete()

    return render(request, 'users/crud-serial-number.html')
```

In this code the meter AND the data will be deleted. It will first search all the data related to the given meter and then delete all the data.

This code will only have an impact if the request method is POST, we use jQuery in our template to make this call:

```
$.post({
    url: "/serialNumbers/DeleteWithData/" + serialID,
    data: {csrfmiddlewaretoken: '{{ csrf_token }}'}
}).done(function () {
    location.reload()
})
```

updateMeterName

This will be triggered when visiting /serialNumber/updateMeterName/(id) and will trigger the following code in the *views.py*:

```
def updateMeterName(request, id):
    serial = SerialNumber.objects.filter(id=id).first()
    if request.method == "POST":
        form = UpdateMeterNameForm(request.POST, instance=serial)
        if form.is_valid():
            serial.save()
            messages.success(request, f'Successfully changed meter name!')
        else:
            messages.error(request, f'Oops, something went wrong!')
    return render(request, 'users/crud-serial-number.html')
```

This code first gets the object by id. Then create a form with the new name. We do this because now we can easily check if it is valid. If it is valid, we save it.

This code will only have an impact if the request method is POST, we use jQuery in our template to make this call:

```
$.post({
    url: "/serialNumbers/updateMeterName/" + serialID,
    data: {csrfmiddlewaretoken: '{{ csrf_token }}', name:name}
}).done(function () {
    location.reload()
})
```

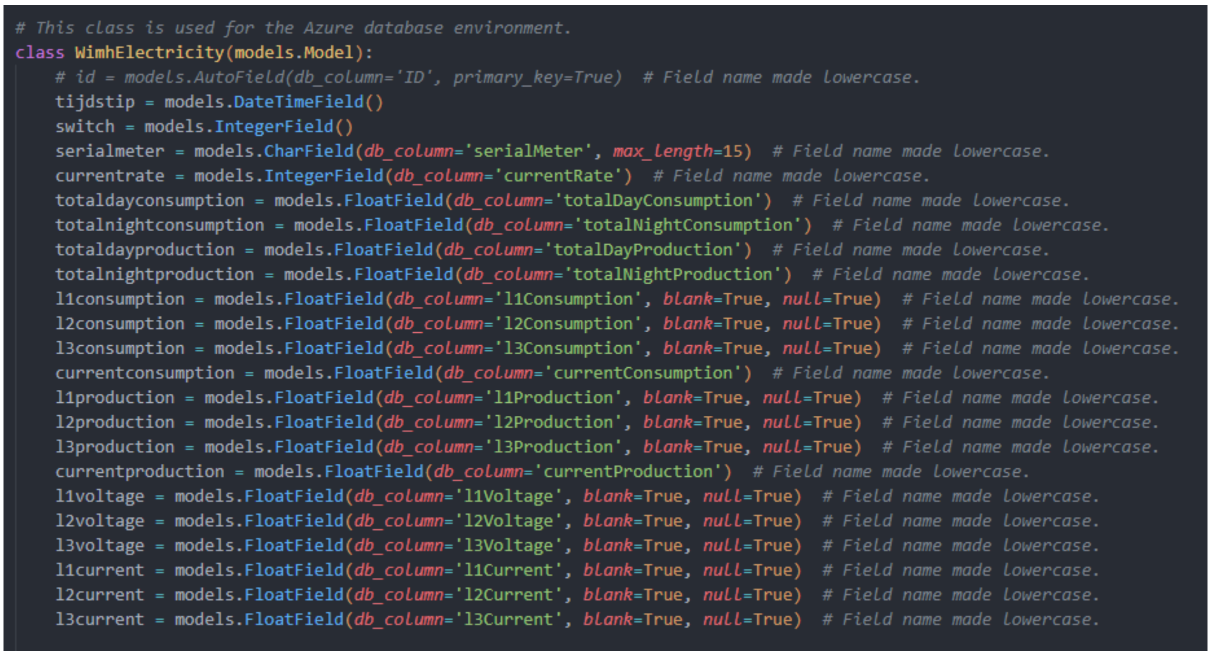
API

Our Raspberry Pi reads the data from the Digital Meter. Now it needs to place this data in the database. Therefore, we have made an API to which the Raspberry Pi can send data.

Everything related to the API can be found in the *api* folder.

Incoming data

Inside *users\models.py* you can find at the bottom of the page the following piece of code:



```python
# This class is used for the Azure database environment.
class WimhElectricity(models.Model):
    # id = models.AutoField(db_column='ID', primary_key=True)  # Field name made lowercase.
    tijdstip = models.DateTimeField()
    switch = models.IntegerField()
    serialmeter = models.CharField(db_column='serialMeter', max_length=15)  # Field name made lowercase.
    currentrate = models.IntegerField(db_column='currentRate')  # Field name made lowercase.
    totaldayconsumption = models.FloatField(db_column='totalDayConsumption')  # Field name made lowercase.
    totalnightconsumption = models.FloatField(db_column='totalNightConsumption')  # Field name made lowercase.
    totaldayproduction = models.FloatField(db_column='totalDayProduction')  # Field name made lowercase.
    totalnightproduction = models.FloatField(db_column='totalNightProduction')  # Field name made lowercase.
    l1consumption = models.FloatField(db_column='l1Consumption', blank=True, null=True)  # Field name made lowercase.
    l2consumption = models.FloatField(db_column='l2Consumption', blank=True, null=True)  # Field name made lowercase.
    l3consumption = models.FloatField(db_column='l3Consumption', blank=True, null=True)  # Field name made lowercase.
    currentconsumption = models.FloatField(db_column='currentConsumption')  # Field name made lowercase.
    l1production = models.FloatField(db_column='l1Production', blank=True, null=True)  # Field name made lowercase.
    l2production = models.FloatField(db_column='l2Production', blank=True, null=True)  # Field name made lowercase.
    l3production = models.FloatField(db_column='l3Production', blank=True, null=True)  # Field name made lowercase.
    currentproduction = models.FloatField(db_column='currentProduction')  # Field name made lowercase.
    l1voltage = models.FloatField(db_column='l1Voltage', blank=True, null=True)  # Field name made lowercase.
    l2voltage = models.FloatField(db_column='l2Voltage', blank=True, null=True)  # Field name made lowercase.
    l3voltage = models.FloatField(db_column='l3Voltage', blank=True, null=True)  # Field name made lowercase.
    l1current = models.FloatField(db_column='l1Current', blank=True, null=True)  # Field name made lowercase.
    l2current = models.FloatField(db_column='l2Current', blank=True, null=True)  # Field name made lowercase.
    l3current = models.FloatField(db_column='l3Current', blank=True, null=True)  # Field name made lowercase.
```

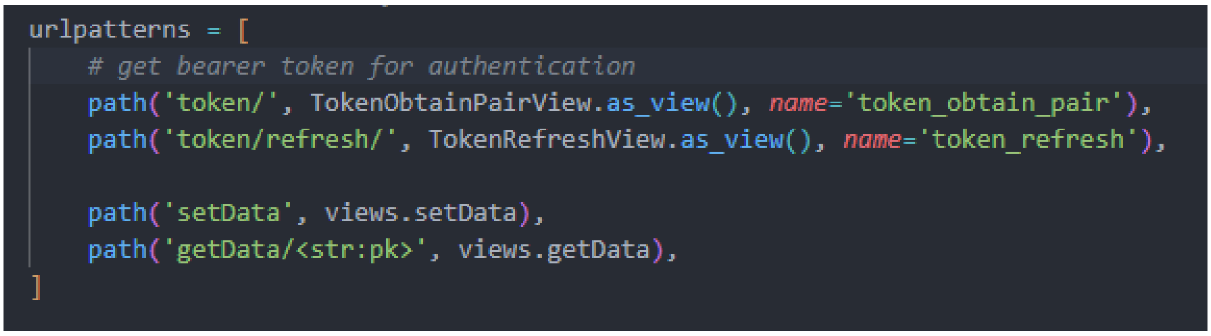
This shows us what the Raspberry Pi will send to us. Note that **not everything** is **required**!

This will be sent in **JSON format**.

Endpoints

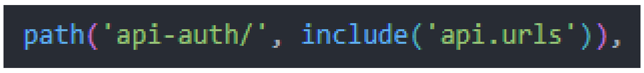
We have 4 endpoints in our API.

Inside the *api/urls.py* you will find the endpoints:



```python
urlpatterns = [
    # get bearer token for authentication
    path('token/', TokenObtainPairView.as_view(), name='token_obtain_pair'),
    path('token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),

    path('setData', views.setData),
    path('getData/<str:pk>', views.getData),
]
```

In the root *urls.py* we have specified the beginning of the API's url:



```python
path('api-auth/', include('api.urls')),
```

This means that the full url = api-auth/token.

Token

Because not everyone is allowed to add data to the database, only admins are allowed to make a request to the setData and getData. Therefore, we need authentication. We use simplejwt (bearer token).

The Raspberry Pi will send data every 15 min. But our tokens only last 5 min. In our case the Raspberry Pi will always request a new token but if we were to decrease the interval, we could use the *api-auth/token/refresh* to add more time to our token's lifespan. But getting a new token every time, the Raspberry Pi needs to send data is perfectly fine.

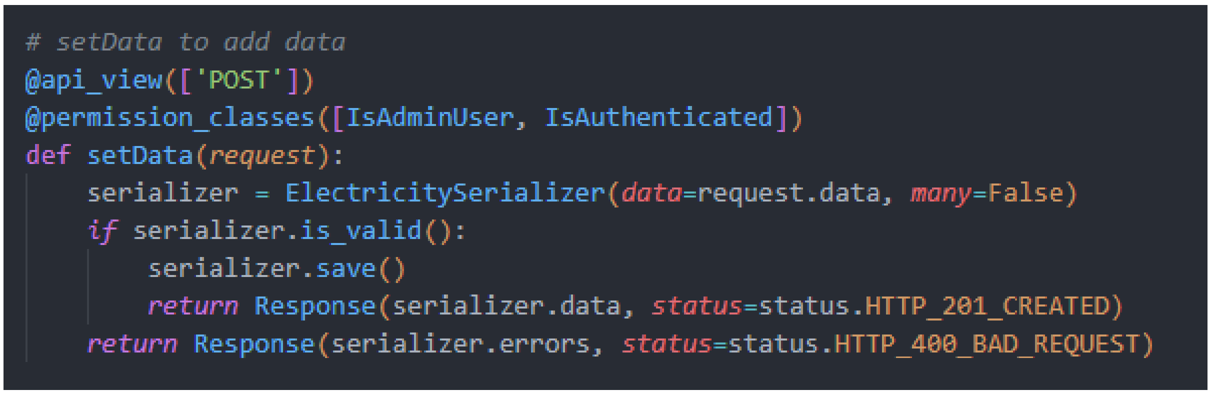
setData

This is the most important endpoint of the API; this will save the data to the database.

Only an admin account can access this endpoint by first making a request to *api-auth/token.* This means that the Raspberry Pi needs an admin account within its script.

Afterwards he needs to add the token to the request as a bearer token. Now it can send data to the API. In the body you need to make a JSON containing the properties listed at the beginning of this chapter. Most of these properties will directly come from the Digital Meter.

Once added, you can send the request and the following code will be executed:



```python
# setData to add data
@api_view(['POST'])
@permission_classes([IsAdminUser, IsAuthenticated])
def setData(request):
    serializer = ElectricitySerializer(data=request.data, many=False)
    if serializer.is_valid():
        serializer.save()
        return Response(serializer.data, status=status.HTTP_201_CREATED)
    return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

In this code you can see that the request needs to be a POST.

The user needs to be authenticated AND needs to be an admin.

If this was successfully, we will first check if the data that has been send matches with the model at the top of this chapter, not everything is required.

If the JSON is valid, the data will be saved and can now be viewed within the application.

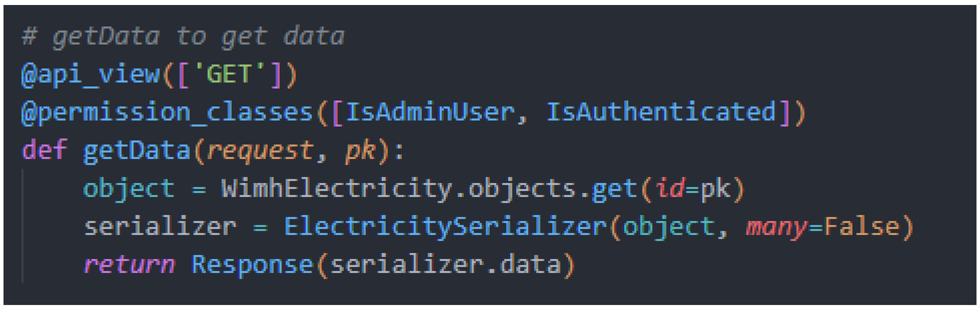
GetData

The getData endpoint is not used within the application, but it is a quick way to check if everything is okay. That is also the reason we left this in the application in case you want to see if the data is still good.

The url for this endpoint = *api-auth/getData/(id)*

This will return one measurement.

If called, the following code will be executed:



```
# getData to get data
@api_view(['GET'])
@permission_classes([IsAdminUser, IsAuthenticated])
def getData(request, pk):
    object = WimhElectricity.objects.get(id=pk)
    serializer = ElectricitySerializer(object, many=False)
    return Response(serializer.data)
```

Here we can see that it is a GET method. And that again we need to be an admin and authenticated.

It will then return the requested data.

Cloud and Security

3IT had some requirements for our project including the cloud provider. We had to use Microsoft Azure. To deploy all the necessary resources on Azure we've used GitLab-CI in combination with Terraform and Docker.

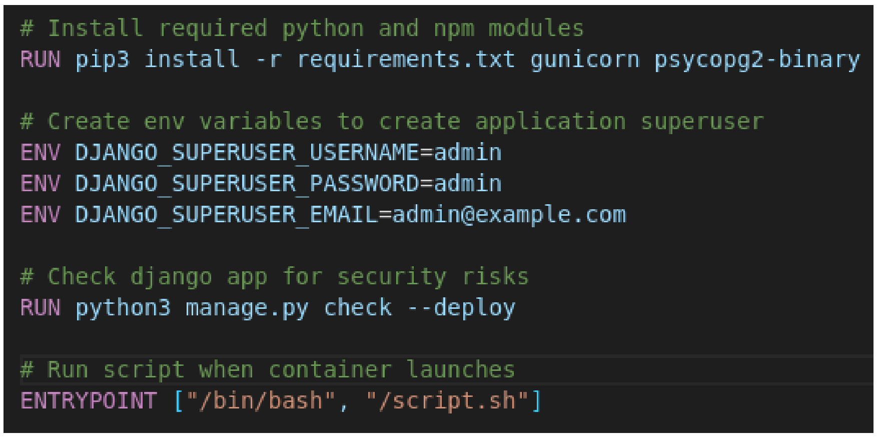
Docker

As mentioned above we chose to deploy our application on Azure App Service. This Azure resource supports multiple types of applications: PHP, Ruby, Node.js, Java, Python, Containers,.... Since we already tested our application in docker before we tried to deploy it in Azure it seemed logical to deploy this docker container to the Azure App Service. This way we have more control of the container itself. We created this docker container from scratch.

Dockerfile

In this dockerfile we started from a default Ubuntu/Nginx image. First, we had to install a few extra necessary packages. After this we copy the needed files and folders over to the image and configure the Nginx and Gunicorn. At startup the container runs a script that keeps running so the container never dies.

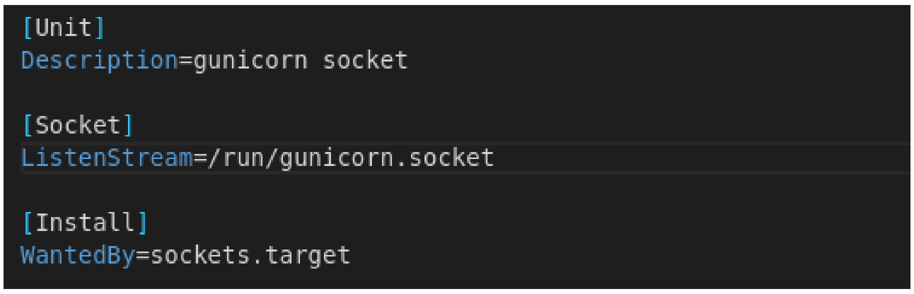
```
# Start from ubuntu/nginx image
FROM ubuntu/nginx:1.18-22.04_beta

# Update all packages and install required packages
RUN apt-get update
RUN apt-get install -y systemctl curl gnupg python3-pip

# Install microsoft odbc sql driver
RUN curl https://packages.microsoft.com/keys/microsoft.asc |
apt-key add -
RUN curl https://packages.microsoft.com/config/ubuntu/22.04/
prod.list > /etc/apt/sources.list.d/mssql-release.list
RUN apt-get update
RUN ACCEPT_EULA=Y apt-get install -y msodbcsql18

# Create application directory and make working directory
RUN mkdir /app
WORKDIR /app

# Copy all the needed files into the docker image
COPY ./energy-application /app
COPY ./gunicorn.socket /etc/systemd/system/gunicorn.socket
COPY ./gunicorn.service /etc/systemd/system/gunicorn.service
COPY ./nginx /etc/nginx/sites-available/energy_application
COPY ./script.sh /script.sh

# Config nginx to run the application
RUN rm /etc/nginx/sites-enabled/default
RUN ln -s /etc/nginx/sites-available/energy_application /etc/
nginx/sites-enabled
```



```
# Install required python and npm modules
RUN pip3 install -r requirements.txt gunicorn psycopg2-binary

# Create env variables to create application superuser
ENV DJANGO_SUPERUSER_USERNAME=admin
ENV DJANGO_SUPERUSER_PASSWORD=admin
ENV DJANGO_SUPERUSER_EMAIL=admin@example.com

# Check django app for security risks
RUN python3 manage.py check --deploy

# Run script when container launches
ENTRYPOINT ["/bin/bash", "/script.sh"]
```

Gunicorn socket

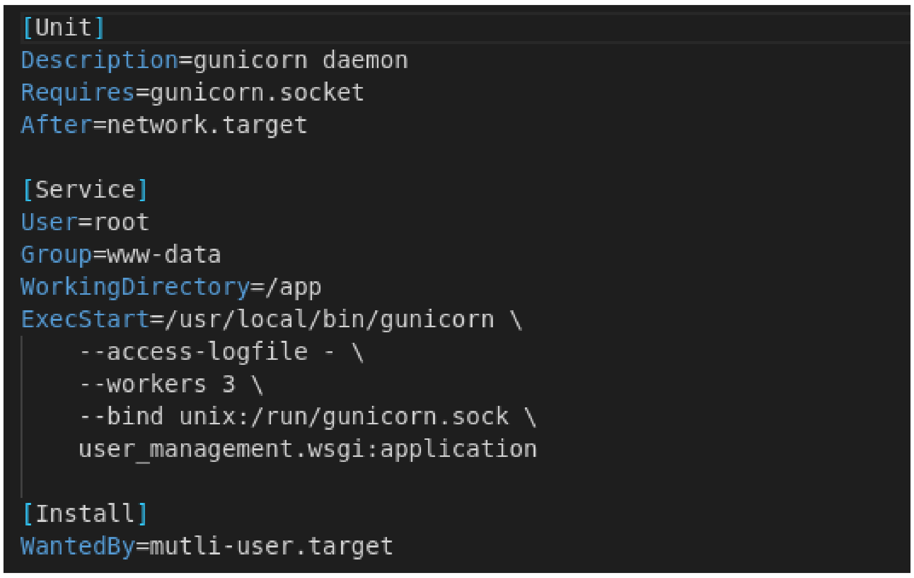
In our docker image we run our Django application in Gunicorn as a generic WSGI application. Of course, this gunicorn needs to be configured. First, we create a socket this socket will serve our dynamic application.



```
[Unit]
Description=gunicorn socket

[Socket]
ListenStream=/run/gunicorn.socket

[Install]
WantedBy=sockets.target
```

Gunicorn service

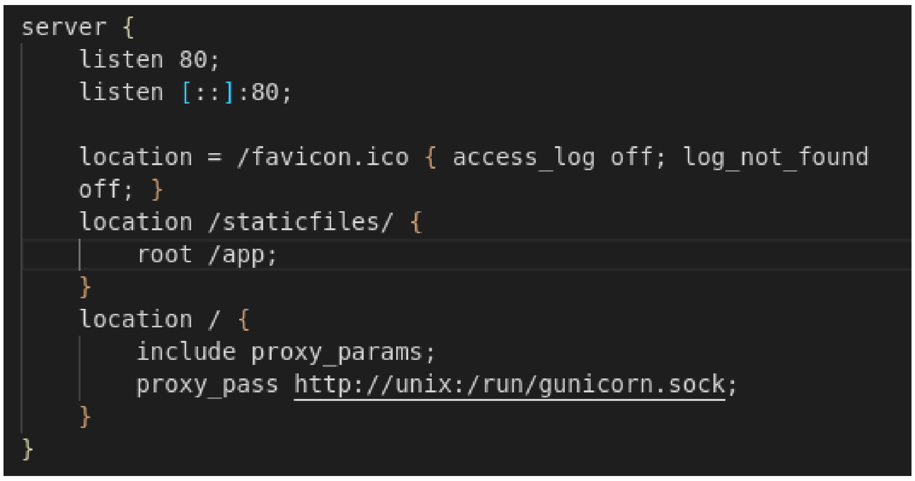
We are using Gunicorn. We just created our Gunicorn Socket. Now we need to create a service that will run use this socket and run it with workers. We have configured 3 workers.



```
[Unit]
Description=gunicorn daemon
Requires=gunicorn.socket
After=network.target

[Service]
User=root
Group=www-data
WorkingDirectory=/app
ExecStart=/usr/local/bin/gunicorn \
    --access-logfile - \
    --workers 3 \
    --bind unix:/run/gunicorn.sock \
    user_management.wsgi:application

[Install]
WantedBy=mutli-user.target
```

Nginx config

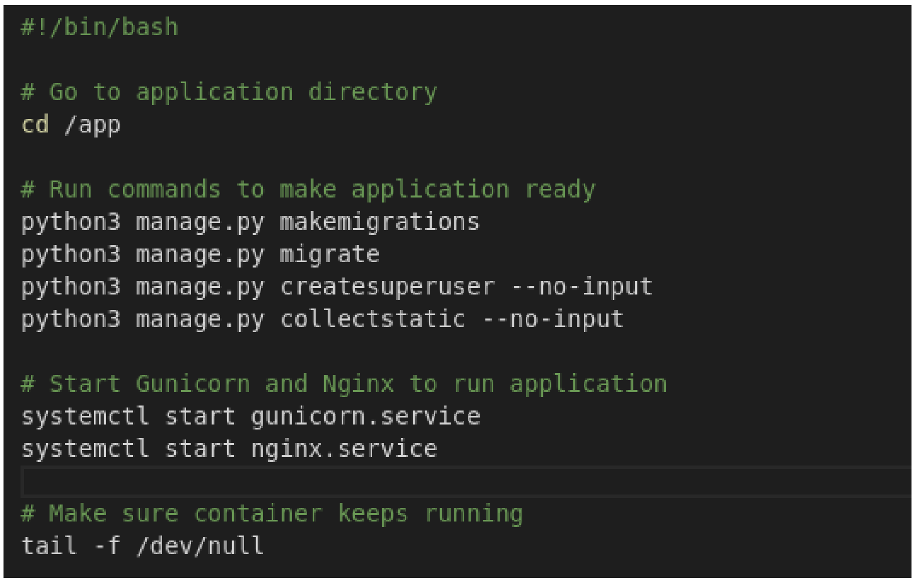
The Nginx service inside our docker container needs a configuration to be to work correctly. This file is copied into the docker image as mentioned above. In this file we configure Nginx to use the Gunicorn socket to serve the application, static files are served directly by the Nginx service itself.



```
server {
    listen 80;
    listen [::]:80;

    location = /favicon.ico { access_log off; log_not_found
    off; }
    location /staticfiles/ {
        root /app;
    }
    location / {
        include proxy_params;
        proxy_pass http://unix:/run/gunicorn.sock;
    }
}
```

Startup script

This startup script creates the database migrations and creates an administrator account in the application. Then the Nginx and Gunicorn services are started.



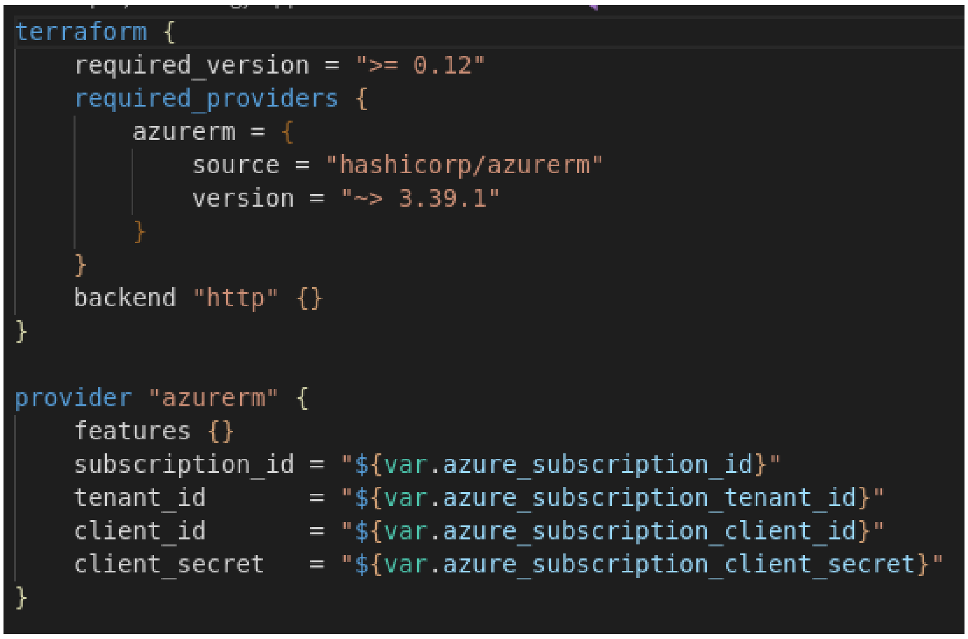
```
#!/bin/bash

# Go to application directory
cd /app

# Run commands to make application ready
python3 manage.py makemigrations
python3 manage.py migrate
python3 manage.py createsuperuser --no-input
python3 manage.py collectstatic --no-input

# Start Gunicorn and Nginx to run application
systemctl start gunicorn.service
systemctl start nginx.service

# Make sure container keeps running
tail -f /dev/null
```

Terraform

Terraform is an open source “Infrastructure as Code” tool, created by HashiCorp. It is a declarative coding tool. Terraform has multiple providers including an Azure provider. With the help of this provider, we can create all the necessary resources on Azure using Terraform.

Provider

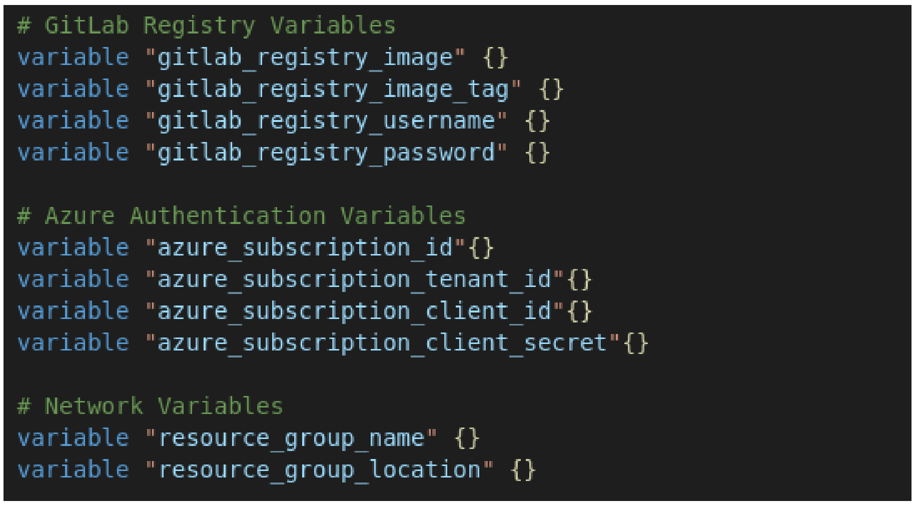
To be able to use Terraform we need to specify which provider we’ll be using. In our case we’re using the azurerm provider from HashiCorp. We need to define our credentials for this provider so Terraform knows which Azure account to create the resources in and to make sure we’re authorised to do so. To make sure our Terraform files keep working we’re specifying the versions of our provider. We’ll also be making use of a http backend to store our Terraform statefile in.



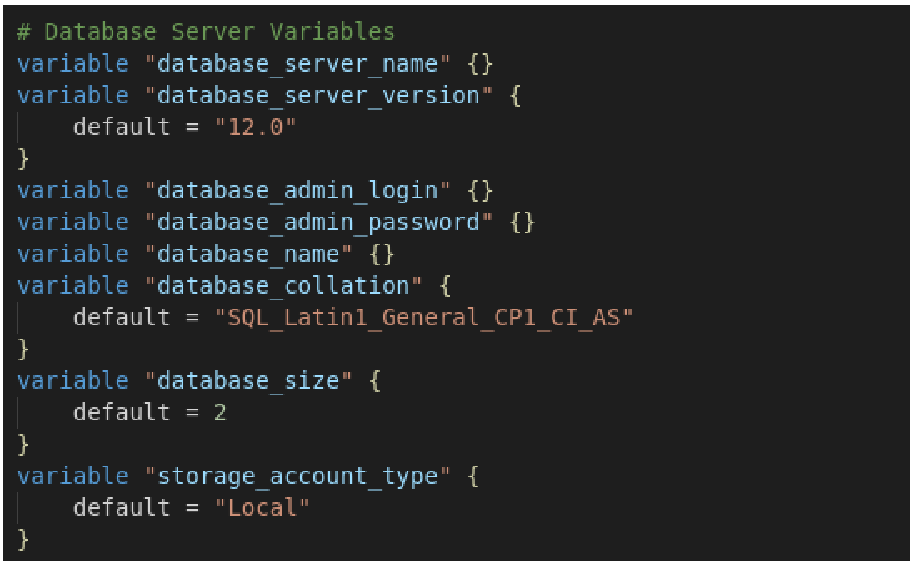
```
terraform {
    required_version = ">= 0.12"
    required_providers {
        azurerm = {
            source = "hashicorp/azurerm"
            version = "~> 3.39.1"
        }
    }
    backend "http" {}
}

provider "azurerm" {
    features {}
    subscription_id = "${var.azure_subscription_id}"
    tenant_id       = "${var.azure_subscription_tenant_id}"
    client_id       = "${var.azure_subscription_client_id}"
    client_secret   = "${var.azure_subscription_client_secret}"
}
```

Variables

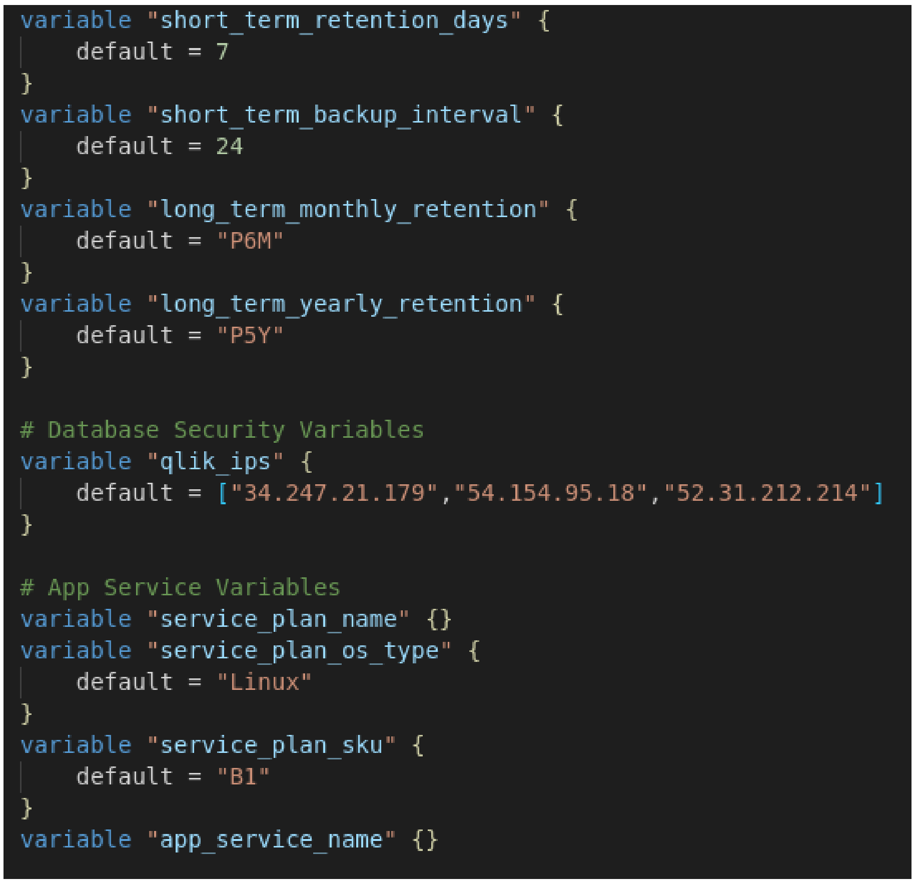
To make the coding easier we’ve user a variables file. This way we only have to declare name and values of a variable once. Most of these variables are imported from the GitLab Vault, but as you can see some have been declared in the file itself. The names and values that need to be confidential or easy to change are stored in the GitLab Vault, the rest is already filled in.



```
# GitLab Registry Variables
variable "gitlab_registry_image" {}
variable "gitlab_registry_image_tag" {}
variable "gitlab_registry_username" {}
variable "gitlab_registry_password" {}

# Azure Authentication Variables
variable "azure_subscription_id"{}
variable "azure_subscription_tenant_id"{}
variable "azure_subscription_client_id"{}
variable "azure_subscription_client_secret"{}

# Network Variables
variable "resource_group_name" {}
variable "resource_group_location" {}
```



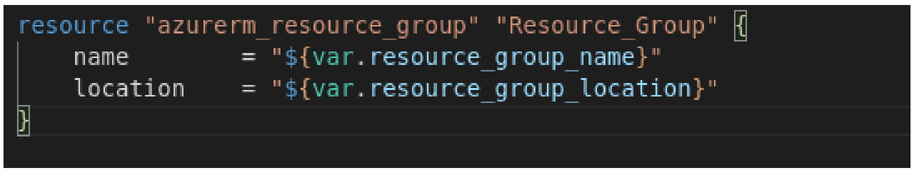
```
# Database Server Variables
variable "database_server_name" {}
variable "database_server_version" {
    default = "12.0"
}
variable "database_admin_login" {}
variable "database_admin_password" {}
variable "database_name" {}
variable "database_collation" {
    default = "SQL_Latin1_General_CP1_CI_AS"
}
variable "database_size" {
    default = 2
}
variable "storage_account_type" {
    default = "Local"
}
```



```
variable "short_term_retention_days" {
    default = 7
}
variable "short_term_backup_interval" {
    default = 24
}
variable "long_term_monthly_retention" {
    default = "P6M"
}
variable "long_term_yearly_retention" {
    default = "P5Y"
}

# Database Security Variables
variable "qlik_ips" {
    default = ["34.247.21.179","54.154.95.18","52.31.212.214"]
}

# App Service Variables
variable "service_plan_name" {}
variable "service_plan_os_type" {
    default = "Linux"
}
variable "service_plan_sku" {
    default = "B1"
}
variable "app_service_name" {}
```

Resource Group

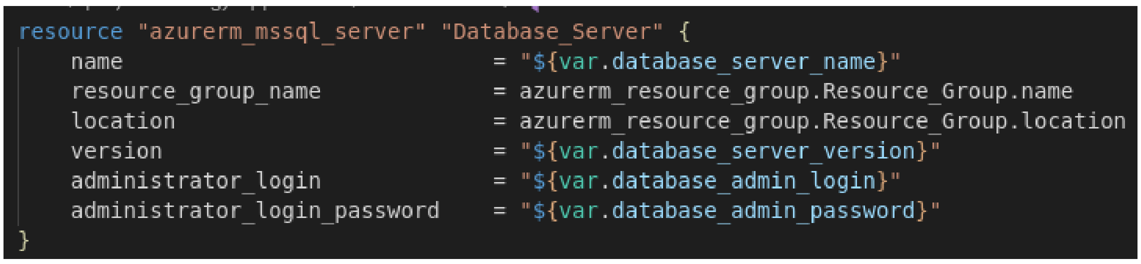
Azure uses resources groups to place resources in a specific location or region. You can name this resource group yourself. All other resources are then placed inside this resource group.



```
resource "azurerm_resource_group" "Resource_Group" {
    name         = "${var.resource_group_name}"
    location     = "${var.resource_group_location}"
}
```

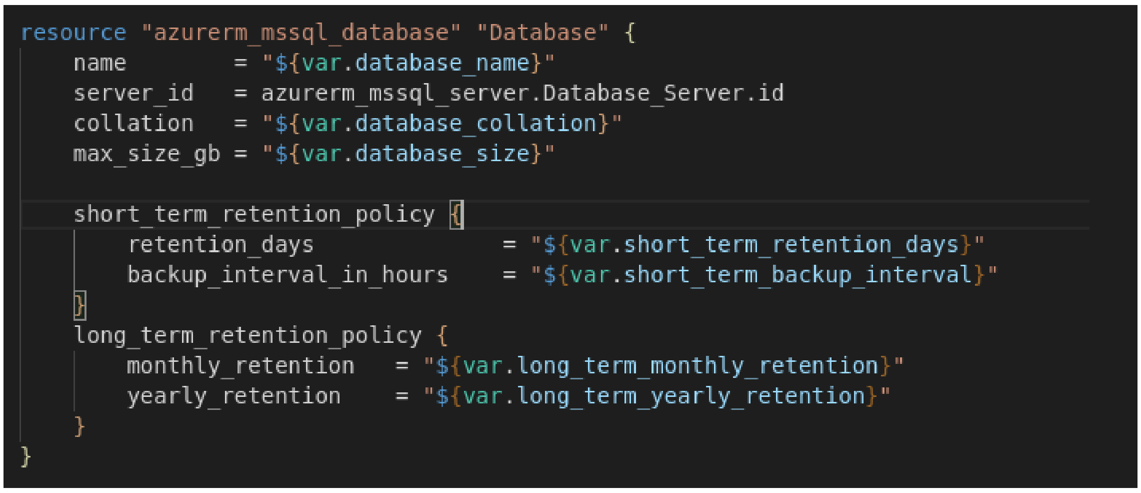
Azure SQL Database

After our variables declaration we can define what our database should like. We previously said that we will be using a MSSQL database on Azure. Before we can create the database itself we need to create a database server first. This server is location in the resource group we created 1 step earlier. We also need to create an administrator account for this server, we'll be using a password with 64 characters to make it secure.



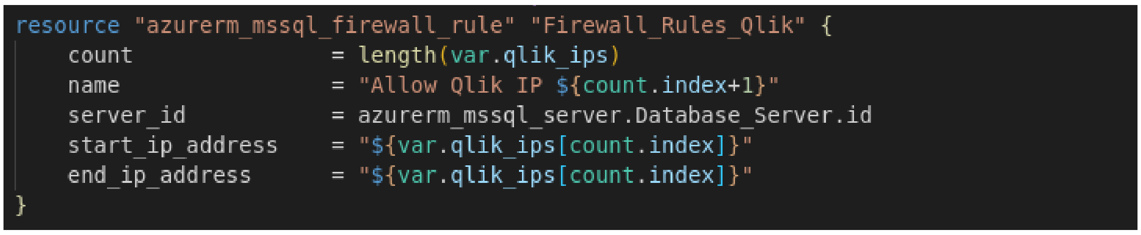
```
resource "azurerm_mssql_server" "Database_Server" {
    name                              = "${var.database_server_name}"
    resource_group_name               = azurerm_resource_group.Resource_Group.name
    location                          = azurerm_resource_group.Resource_Group.location
    version                           = "${var.database_server_version}"
    administrator_login               = "${var.database_admin_login}"
    administrator_login_password      = "${var.database_admin_password}"
}
```

Our server is declared, now it's the turn of the database. Because we will only be gathering numerical data, and not that much either, a 2GB database should be large enough to handle our application for the time being. Of course we need to set the retention periods. For our point-in-time-backup we'll take a 7-day retention period that will create an incremental backup every 24 hours. Our long-term backups will be kept longer. Our monthly backup will be kept for 6 months, and our yearly backup will be kept for 5 years.



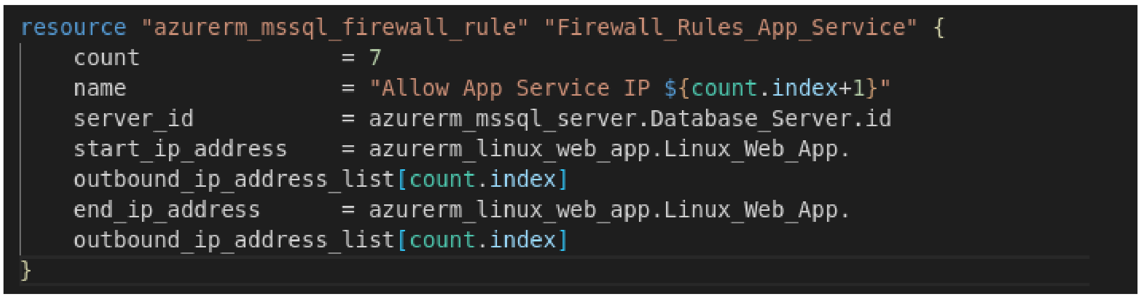
```
resource "azurerm_mssql_database" "Database" {
    name        = "${var.database_name}"
    server_id   = azurerm_mssql_server.Database_Server.id
    collation   = "${var.database_collation}"
    max_size_gb = "${var.database_size}"

    short_term_retention_policy {
        retention_days                = "${var.short_term_retention_days}"
        backup_interval_in_hours      = "${var.short_term_backup_interval}"
    }
    long_term_retention_policy {
        monthly_retention    = "${var.long_term_monthly_retention}"
        yearly_retention     = "${var.long_term_yearly_retention}"
    }
}
```

After the database declaration we need to make a few firewall rules to make sure some resources or services are allowed to make a connection to our database. Our visualisation software needs access to our database so we’ve created firewall rules based on a list with IP addresses that we now are for Qlik.



```
resource "azurerm_mssql_firewall_rule" "Firewall_Rules_Qlik" {
    count                   = length(var.qlik_ips)
    name                    = "Allow Qlik IP ${count.index+1}"
    server_id               = azurerm_mssql_server.Database_Server.id
    start_ip_address        = "${var.qlik_ips[count.index]}"
    end_ip_address          = "${var.qlik_ips[count.index]}"
}
```

Our app service we’ll define in the next step also needs to have access to our database because the API our Raspberry Pi uses is housed inside this app service. After the creation of our app service, we’ll ask for a list of IP addresses in use by our app service. Based upon this list we’ll create firewall rules like we did for Qlik.



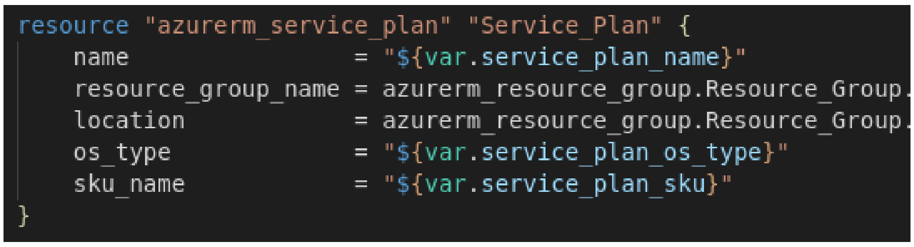
```
resource "azurerm_mssql_firewall_rule" "Firewall_Rules_App_Service" {
    count                   = 7
    name                    = "Allow App Service IP ${count.index+1}"
    server_id               = azurerm_mssql_server.Database_Server.id
    start_ip_address        = azurerm_linux_web_app.Linux_Web_App.
    outbound_ip_address_list[count.index]
    end_ip_address          = azurerm_linux_web_app.Linux_Web_App.
    outbound_ip_address_list[count.index]
}
```

Azure App Service

Now that our database has been defined, we can configure our application itself. As mentioned above we’ve been using the Azure App Service. With it you can quickly deploy and manage applications in Azure and make them available on the internet without the need to deploy all the network infrastructure yourself.

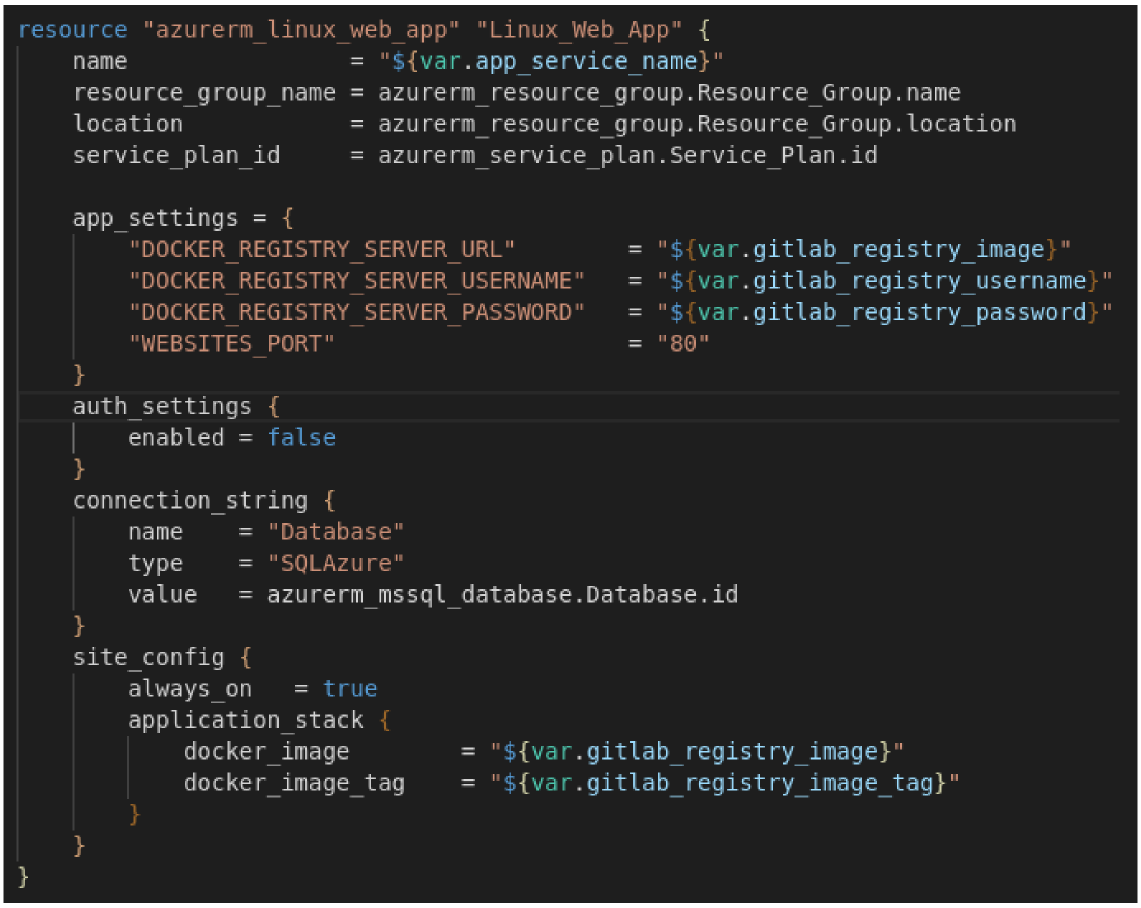
Our Azure App Service consists of a service plan and a Linux web app. The application is then deployed on this service plan. We created our service plan and Linux web app inside of the previously created resource group.

We need to declare what OS our application will be using by configuring it in the service plan. The size and performance of our application also needs to be specified by giving a SKU-name.



```
resource "azurerm_service_plan" "Service_Plan" {
    name                = "${var.service_plan_name}"
    resource_group_name = azurerm_resource_group.Resource_Group.
    location            = azurerm_resource_group.Resource_Group.
    os_type             = "${var.service_plan_os_type}"
    sku_name            = "${var.service_plan_sku}"
}
```

When our service plan is created, we can create our Linux web app on top of our service plan. As we said in the part about docker we will be using a custom docker image. The Linux web app needs to know where to fetch this image and what credentials are needed to fetch it. As we will be using a database that needs a connection to our application, we need to create a connection string as well.



```
resource "azurerm_linux_web_app" "Linux_Web_App" {
    name                = "${var.app_service_name}"
    resource_group_name = azurerm_resource_group.Resource_Group.name
    location            = azurerm_resource_group.Resource_Group.location
    service_plan_id     = azurerm_service_plan.Service_Plan.id

    app_settings = {
        "DOCKER_REGISTRY_SERVER_URL"        = "${var.gitlab_registry_image}"
        "DOCKER_REGISTRY_SERVER_USERNAME"   = "${var.gitlab_registry_username}"
        "DOCKER_REGISTRY_SERVER_PASSWORD"   = "${var.gitlab_registry_password}"
        "WEBSITES_PORT"                     = "80"
    }
    auth_settings {
        enabled = false
    }
    connection_string {
        name     = "Database"
        type     = "SQLAzure"
        value    = azurerm_mssql_database.Database.id
    }
    site_config {
        always_on    = true
        application_stack {
            docker_image        = "${var.gitlab_registry_image}"
            docker_image_tag    = "${var.gitlab_registry_image_tag}"
        }
    }
}
```

GitLab-CI

GitLab CI/CD is a tool for software development using the continuous methodologies. As we like to automate our deployment of our application and our database we'll be using GitLab-CI. When using GitLab-CI we first need to create a .gitlab-ci.yml file wherein we define what needs to happen during our automated pipeline.

GitLab Vault

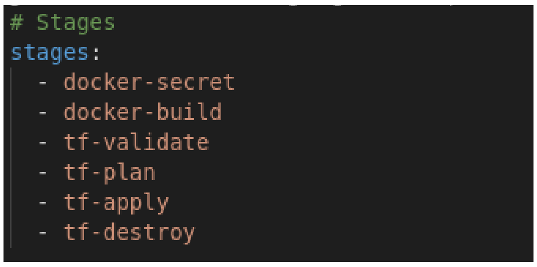
Before we can use the GitLab-CI and create our .gitlab-ci.yml we first need to declare all the variables used.

Here is a list of all the variables we need to define:

- The following can be obtained from Azure:
    - AZURE_SUBSCRIPTION_ID
    - AZURE_SUBSCRIPTION_TENANT_ID
    - AZURE_SUBSCRIPTION_CLIENT_ID
    - AZURE_SUBSCRIPTION_CLIENT_SECRET
- These variables need to be gathered from GitLab itself:
    - GITLAB_USERNAME
    - GITLAB_ACCESS_TOKEN (Access Token with API permissions)
    - GITLAB_REGISTRY_USERNAME (Deploy Token with read_repository + read_registry permissions)
    - GITLAB_REGISTRY_PASSWORD
- The following variables are needed by Terraform:
    - RESOURCE_GROUP_NAME
    - RESOURCE_GROUP_LOCATION
    - APP_SERVICE_NAME
    - APP_SERVICE_PLAN_NAME
    - DATABASE_NAME
    - DATABASE_SERVER_NAME
    - DATABASE_ADMIN_LOGIN
    - DATABASE_ADMIN_PASSWORD
- Application variables:
    - APPLICATION_SUPERUSER_LOGIN (Administrator account)
    - APPLICATION_SUPERUSER_PASSWORD
    - APPLICATION_SUPERUSER_EMAIL
    - DJANGO_SECRET_KEY

Stages

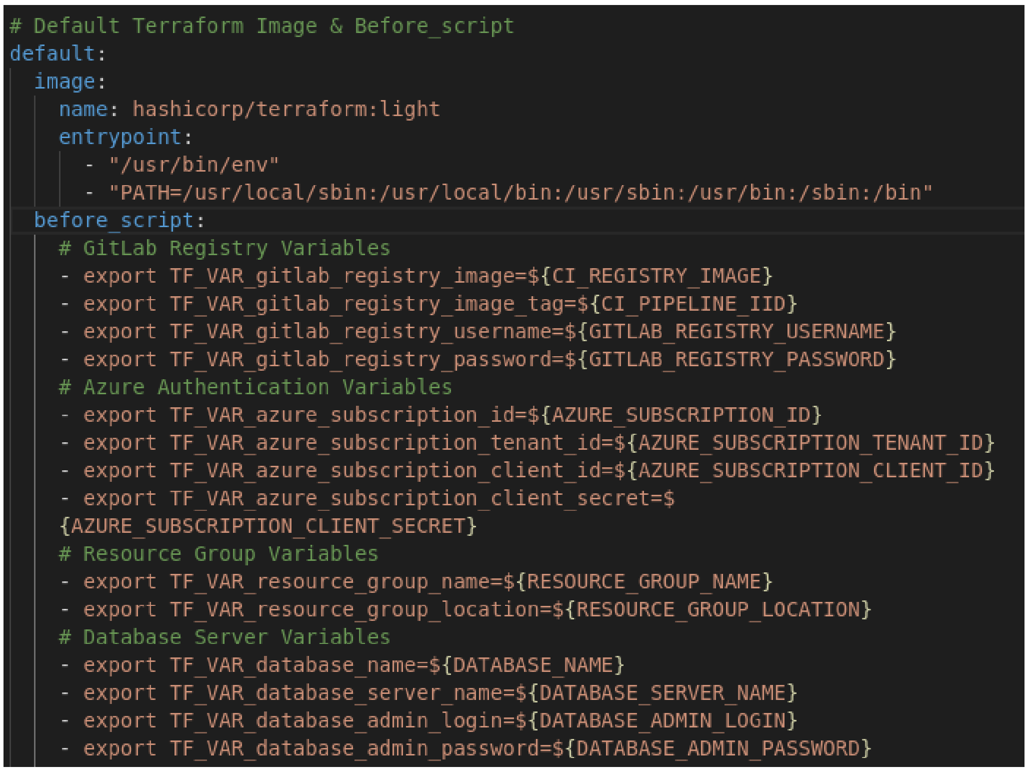
In this .gitlab-ci.yml file we first defined 6 stages. These 6 stages will be executed in order. As you can see the first 2 stages are there to do docker related things. The other 4 stages are related to Terraform.



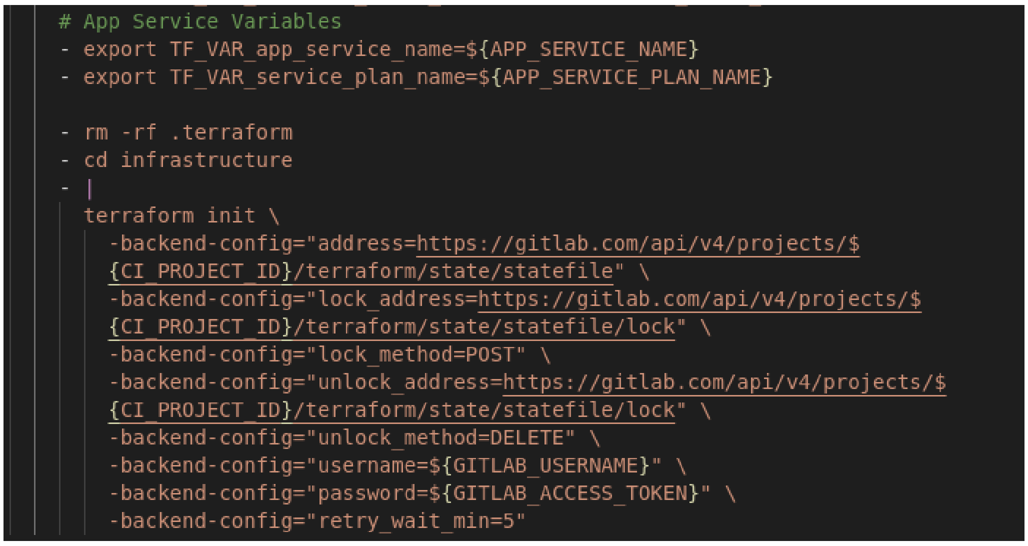
```yaml
# Stages
stages:
  - docker-secret
  - docker-build
  - tf-validate
  - tf-plan
  - tf-apply
  - tf-destroy
```

Defaults

After the stage declaration we declared some defaults including a default image to use and a default before script to use. These default functions are only used by the 4 Terraform related stages. In these functions we define which image is used for the Terraform stages and in the before script we declare all variables that are needed by Terraform. The values are imported from the GitLab Vault to make sure no credentials are stored hard coded in the code itself. At the end of the default before script we define the location to store the Terraform statefile in. This file keeps track of every resource that is created and currently present on Azure.



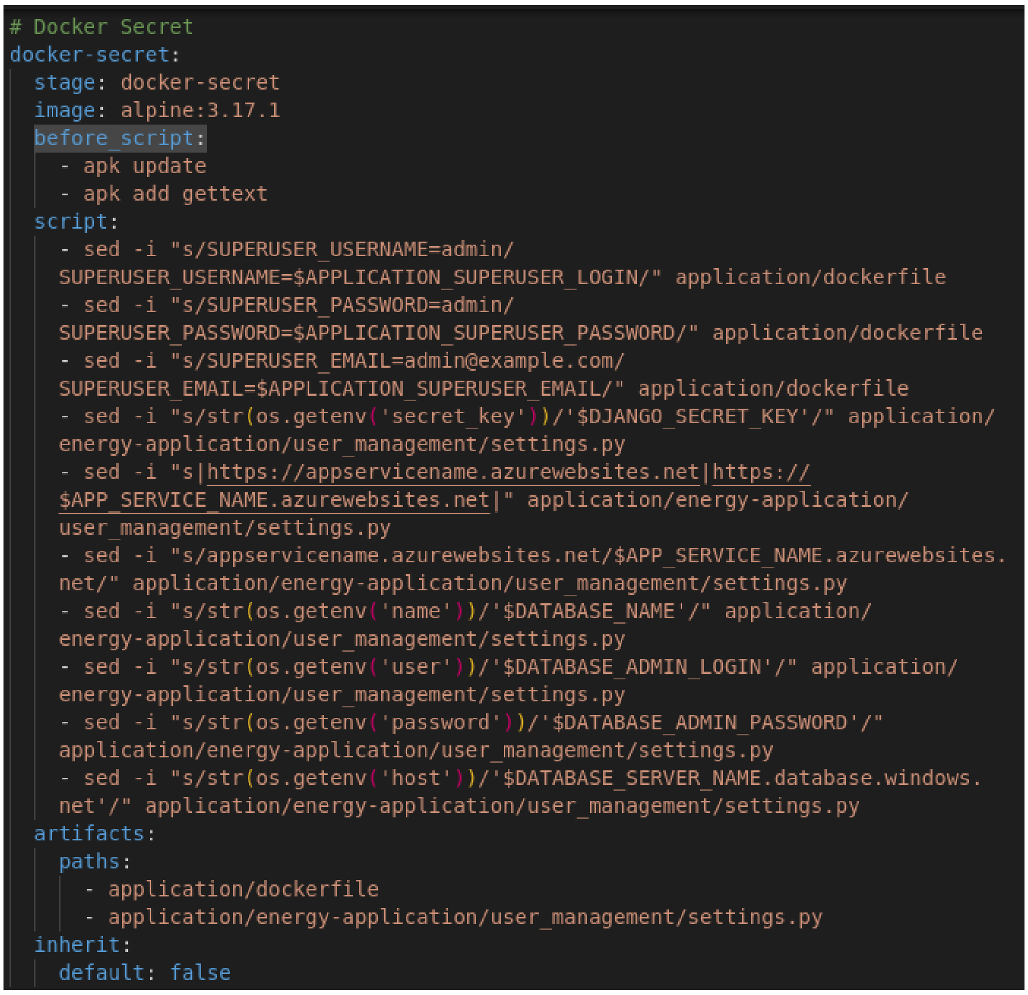
```yaml
# Default Terraform Image & Before_script
default:
  image:
    name: hashicorp/terraform:light
    entrypoint:
      - "/usr/bin/env"
      - "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
  before_script:
    # GitLab Registry Variables
    - export TF_VAR_gitlab_registry_image=${CI_REGISTRY_IMAGE}
    - export TF_VAR_gitlab_registry_image_tag=${CI_PIPELINE_IID}
    - export TF_VAR_gitlab_registry_username=${GITLAB_REGISTRY_USERNAME}
    - export TF_VAR_gitlab_registry_password=${GITLAB_REGISTRY_PASSWORD}
    # Azure Authentication Variables
    - export TF_VAR_azure_subscription_id=${AZURE_SUBSCRIPTION_ID}
    - export TF_VAR_azure_subscription_tenant_id=${AZURE_SUBSCRIPTION_TENANT_ID}
    - export TF_VAR_azure_subscription_client_id=${AZURE_SUBSCRIPTION_CLIENT_ID}
    - export TF_VAR_azure_subscription_client_secret=$
    {AZURE_SUBSCRIPTION_CLIENT_SECRET}
    # Resource Group Variables
    - export TF_VAR_resource_group_name=${RESOURCE_GROUP_NAME}
    - export TF_VAR_resource_group_location=${RESOURCE_GROUP_LOCATION}
    # Database Server Variables
    - export TF_VAR_database_name=${DATABASE_NAME}
    - export TF_VAR_database_server_name=${DATABASE_SERVER_NAME}
    - export TF_VAR_database_admin_login=${DATABASE_ADMIN_LOGIN}
    - export TF_VAR_database_admin_password=${DATABASE_ADMIN_PASSWORD}
```



```
    # App Service Variables
    - export TF_VAR_app_service_name=${APP_SERVICE_NAME}
    - export TF_VAR_service_plan_name=${APP_SERVICE_PLAN_NAME}

    - rm -rf .terraform
    - cd infrastructure
    - |
      terraform init \
        -backend-config="address=https://gitlab.com/api/v4/projects/$
        {CI_PROJECT_ID}/terraform/state/statefile" \
        -backend-config="lock_address=https://gitlab.com/api/v4/projects/$
        {CI_PROJECT_ID}/terraform/state/statefile/lock" \
        -backend-config="lock_method=POST" \
        -backend-config="unlock_address=https://gitlab.com/api/v4/projects/$
        {CI_PROJECT_ID}/terraform/state/statefile/lock" \
        -backend-config="unlock_method=DELETE" \
        -backend-config="username=${GITLAB_USERNAME}" \
        -backend-config="password=${GITLAB_ACCESS_TOKEN}" \
        -backend-config="retry_wait_min=5"
```

Stage 1: docker-secret

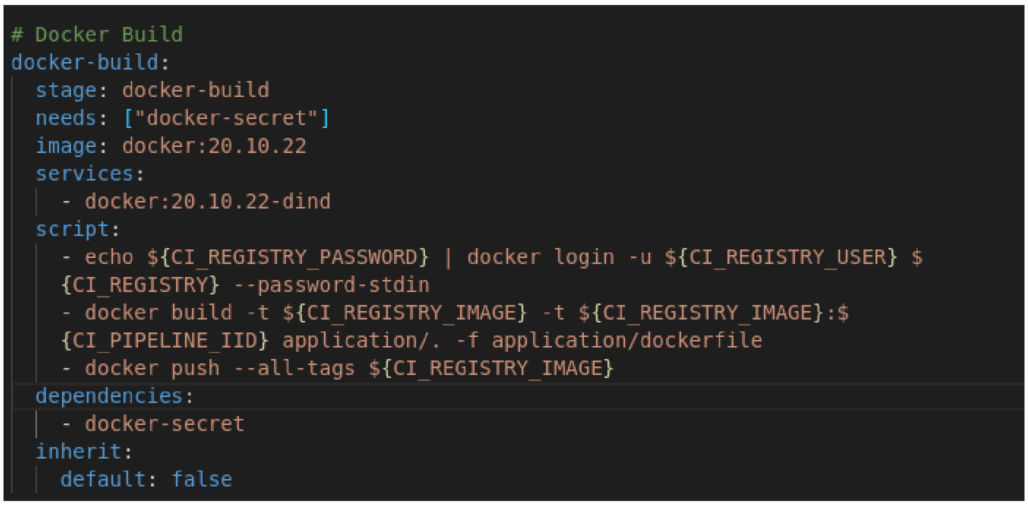
In the first docker stage we prepare our files for the docker build stage. Some variables and credentials in files use in our docker image need to be changes to the values stored in the GitLab Vault. The files that are changes are then kept so they can be used by another stage.



```
# Docker Secret
docker-secret:
  stage: docker-secret
  image: alpine:3.17.1
  before_script:
    - apk update
    - apk add gettext
  script:
    - sed -i "s/SUPERUSER_USERNAME=admin/
    SUPERUSER_USERNAME=$APPLICATION_SUPERUSER_LOGIN/" application/dockerfile
    - sed -i "s/SUPERUSER_PASSWORD=admin/
    SUPERUSER_PASSWORD=$APPLICATION_SUPERUSER_PASSWORD/" application/dockerfile
    - sed -i "s/SUPERUSER_EMAIL=admin@example.com/
    SUPERUSER_EMAIL=$APPLICATION_SUPERUSER_EMAIL/" application/dockerfile
    - sed -i "s/str(os.getenv('secret_key'))/'$DJANGO_SECRET_KEY'/" application/
    energy-application/user_management/settings.py
    - sed -i "s|https://appservicename.azurewebsites.net|https://
    $APP_SERVICE_NAME.azurewebsites.net|" application/energy-application/
    user_management/settings.py
    - sed -i "s/appservicename.azurewebsites.net/$APP_SERVICE_NAME.azurewebsites.
    net/" application/energy-application/user_management/settings.py
    - sed -i "s/str(os.getenv('name'))/'$DATABASE_NAME'/" application/
    energy-application/user_management/settings.py
    - sed -i "s/str(os.getenv('user'))/'$DATABASE_ADMIN_LOGIN'/" application/
    energy-application/user_management/settings.py
    - sed -i "s/str(os.getenv('password'))/'$DATABASE_ADMIN_PASSWORD'/"
    application/energy-application/user_management/settings.py
    - sed -i "s/str(os.getenv('host'))/'$DATABASE_SERVER_NAME.database.windows.
    net'/" application/energy-application/user_management/settings.py
  artifacts:
    paths:
      - application/dockerfile
      - application/energy-application/user_management/settings.py
  inherit:
    default: false
```

Stage 2: docker-build

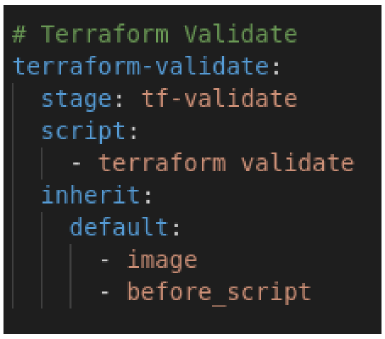
After the docker-secret stage it's time to run the next stage. This stage depends on the last 1. In the docker build stage the files that we're changes in the previous stage are fetch and then the docker build command is used to build the image based on the dockerfile. After the image is build the image will be pushed to the GitLab container registry with the necessary tag. The tag that is used is the number of the pipeline id.



```yaml
# Docker Build
docker-build:
  stage: docker-build
  needs: ["docker-secret"]
  image: docker:20.10.22
  services:
    - docker:20.10.22-dind
  script:
    - echo ${CI_REGISTRY_PASSWORD} | docker login -u ${CI_REGISTRY_USER} $
    {CI_REGISTRY} --password-stdin
    - docker build -t ${CI_REGISTRY_IMAGE} -t ${CI_REGISTRY_IMAGE}:$
    {CI_PIPELINE_IID} application/. -f application/dockerfile
    - docker push --all-tags ${CI_REGISTRY_IMAGE}
  dependencies:
    - docker-secret
  inherit:
    default: false
```

Stage 3: tf-validate

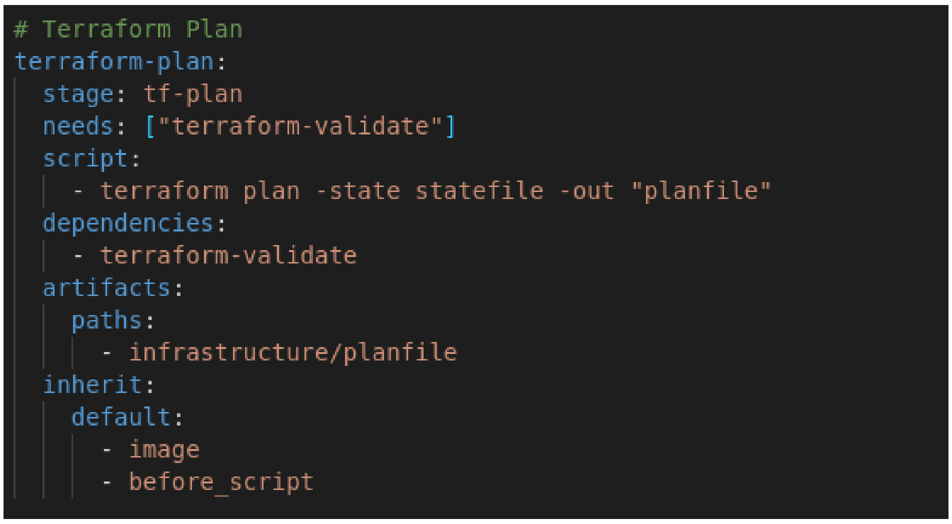
On competition of the last stage the tf-validate stage starts. This stage uses the default image and default before script to make the environment ready for the next commands. In this stage we validate the syntax in all the Terraform files to make sure there are no errors present in the code.



```yaml
# Terraform Validate
terraform-validate:
  stage: tf-validate
  script:
    - terraform validate
  inherit:
    default:
      - image
      - before_script
```

Stage 4: tf-plan

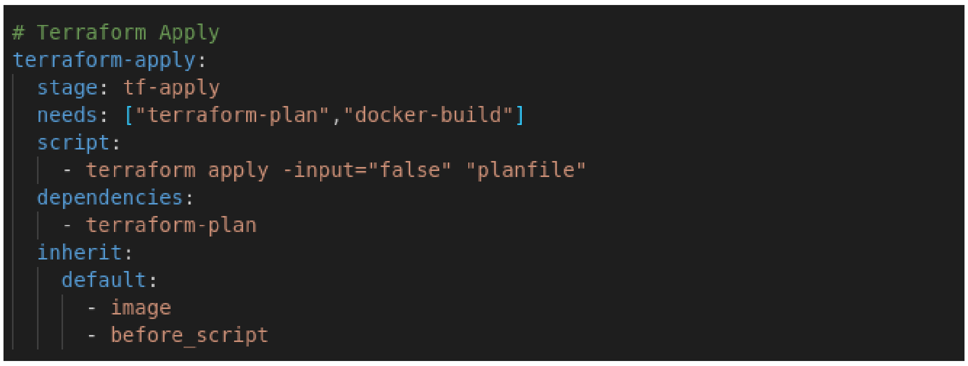
The plan stage comes after the validate stage. In the plan stage the Terraform code is reviewed, and a planfile is being made based on the resources defined in the Terraform files. The plan is then compared with the Terraform statefile stored in GitLab to see which changes still need to be made. This planfile is then saved to be used by the next stage.



```yaml
# Terraform Plan
terraform-plan:
  stage: tf-plan
  needs: ["terraform-validate"]
  script:
    - terraform plan -state statefile -out "planfile"
  dependencies:
    - terraform-validate
  artifacts:
    paths:
      - infrastructure/planfile
  inherit:
    default:
      - image
      - before_script
```

Stage 6: tf-apply

In the tf-apply stage the planfile is fetched and the changes are then being applied on Azure. After completion the statefile in GitLab is update to the newest state. This is also the last stage that is being run automatically.

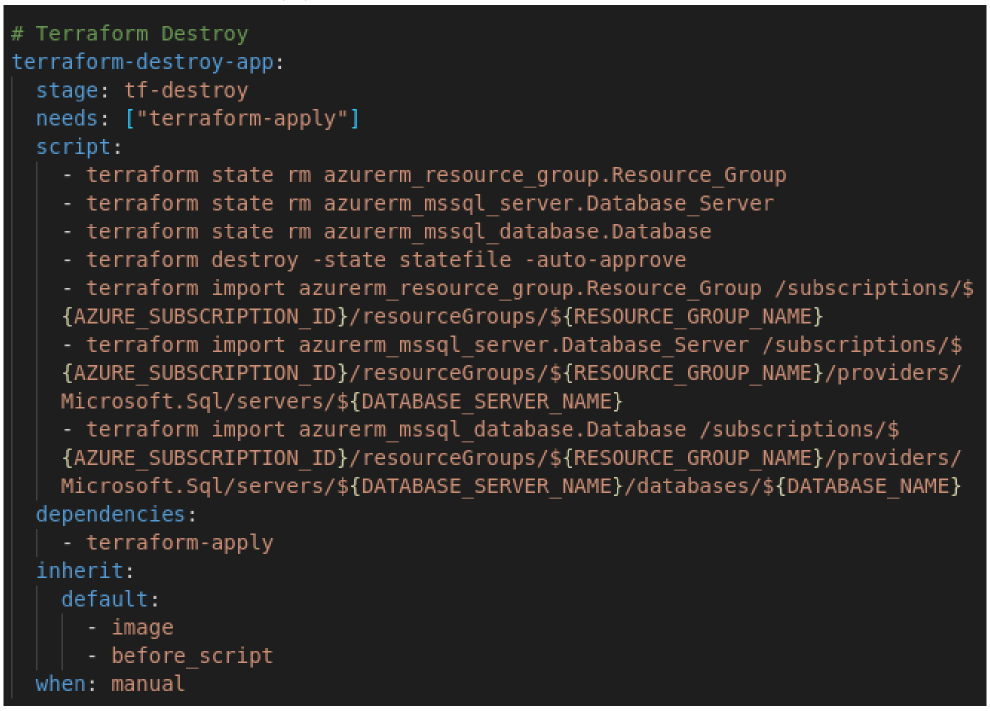


```yaml
# Terraform Apply
terraform-apply:
  stage: tf-apply
  needs: ["terraform-plan","docker-build"]
  script:
    - terraform apply -input="false" "planfile"
  dependencies:
    - terraform-plan
  inherit:
    default:
      - image
      - before_script
```

Stage 7: tf-destroy

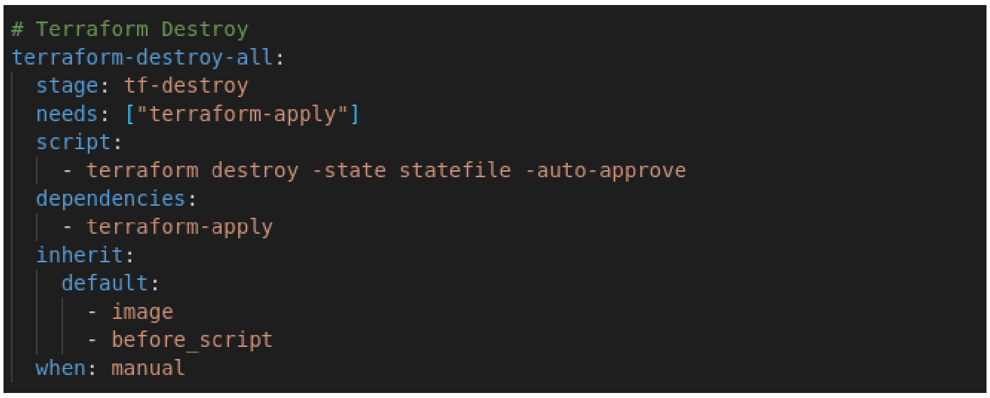
The name tf-destroy says it all. This stage is used to destroy the infrastructure present on Azure. But there are 2 different ways to do this. We have an option to only destroy the application and keep the database and database server and resource group intact and running. The other option destroys everything in Azure. The way we accomplish these 2 options is in the following way.

- In option 1 we first remove the database, database server and resource group from the Terraform statefile so there is no record of its existence in GitLab. Then we run the Terraform destroy command to destroy everything defined in the statefile. When the destroy is complete we again add the remove resources in the statefile to make sure their existence is known if we run a new pipeline



```
# Terraform Destroy
terraform-destroy-app:
  stage: tf-destroy
  needs: ["terraform-apply"]
  script:
    - terraform state rm azurerm_resource_group.Resource_Group
    - terraform state rm azurerm_mssql_server.Database_Server
    - terraform state rm azurerm_mssql_database.Database
    - terraform destroy -state statefile -auto-approve
    - terraform import azurerm_resource_group.Resource_Group /subscriptions/$
    {AZURE_SUBSCRIPTION_ID}/resourceGroups/${RESOURCE_GROUP_NAME}
    - terraform import azurerm_mssql_server.Database_Server /subscriptions/$
    {AZURE_SUBSCRIPTION_ID}/resourceGroups/${RESOURCE_GROUP_NAME}/providers/
    Microsoft.Sql/servers/${DATABASE_SERVER_NAME}
    - terraform import azurerm_mssql_database.Database /subscriptions/$
    {AZURE_SUBSCRIPTION_ID}/resourceGroups/${RESOURCE_GROUP_NAME}/providers/
    Microsoft.Sql/servers/${DATABASE_SERVER_NAME}/databases/${DATABASE_NAME}
  dependencies:
    - terraform-apply
  inherit:
    default:
      - image
      - before_script
  when: manual
```

- In option 2 we directly run the Terraform destroy command to destroy everything in the statefile (which includes everything). The backups of the database however are kept and still stored in Azure.



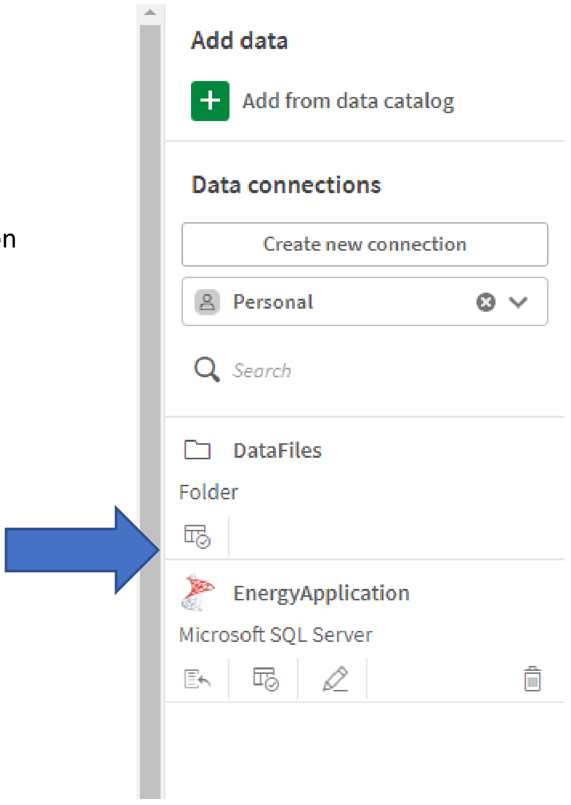
```
# Terraform Destroy
terraform-destroy-all:
  stage: tf-destroy
  needs: ["terraform-apply"]
  script:
    - terraform destroy -state statefile -auto-approve
  dependencies:
    - terraform-apply
  inherit:
    default:
      - image
      - before_script
  when: manual
```

Business Intelligence

Loading the Data

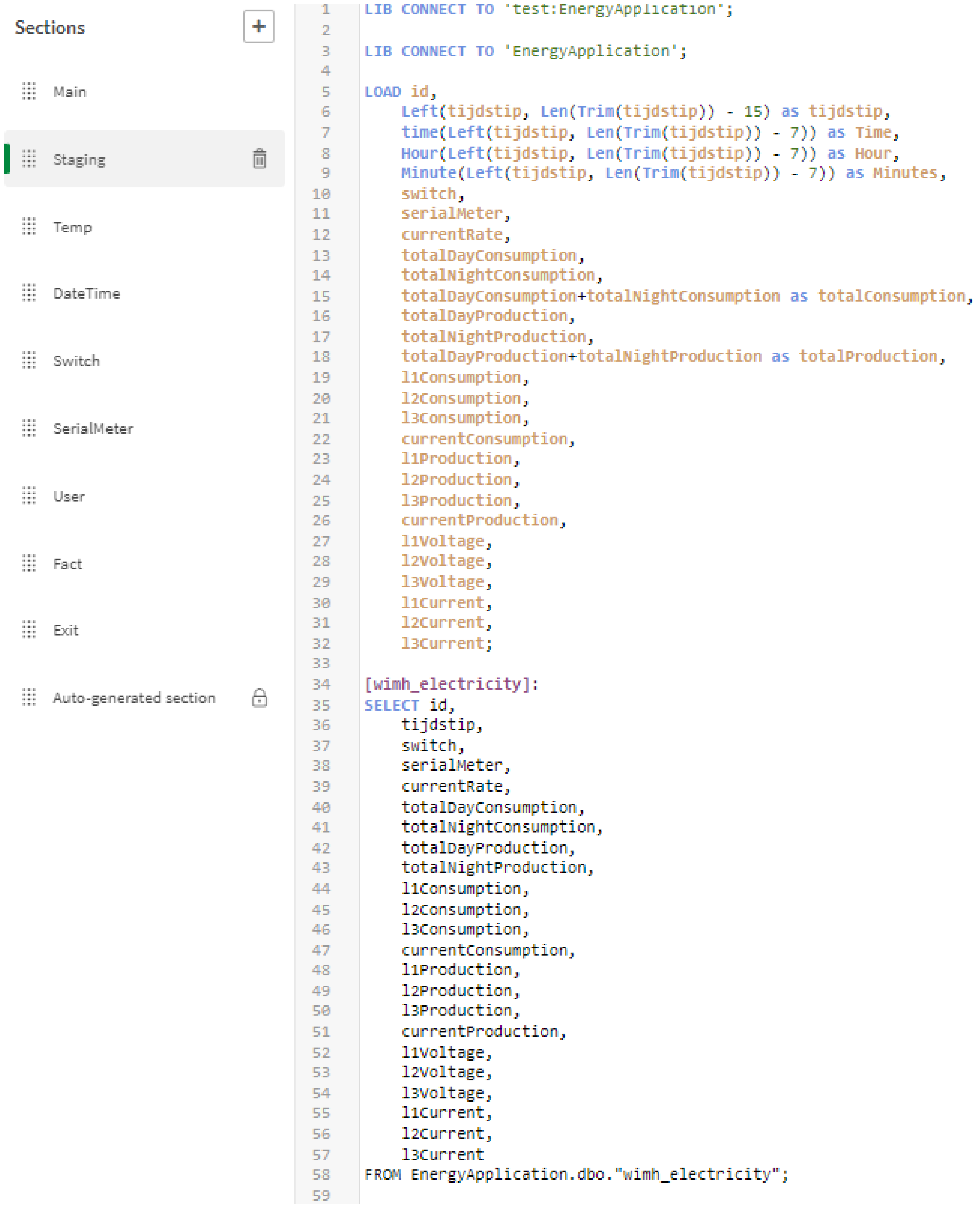
To load the data into our visualizations, we must first find where our database is located. In our situation it is an Azure SQL Database which we can see here.

We ask our security teammate what the credentials of the database are and test our connection. If our connection succeeds, we can load the data and start transforming it to make it easier to create the final product. If our connection fails, we must look at the error which usually is a network problem or an IP-address that is not whitelisted on the database server.

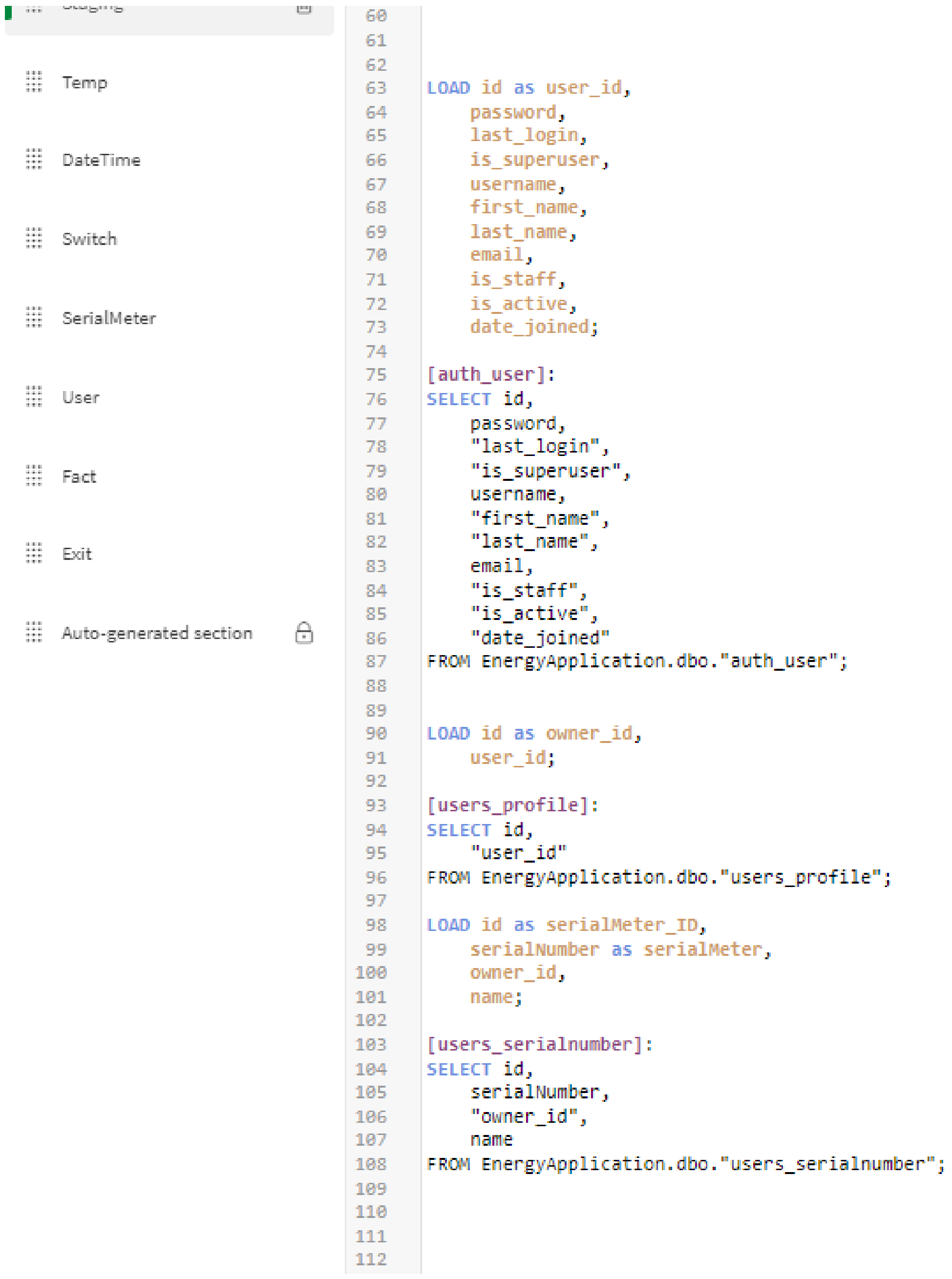


When we start making a script to load our data into the data frame and the dashboards. We made some changes to the timestamp variables so we can work with different intervals and made some extra variables so that we could connect all our tables the way we intended.

Our first section is simply getting our data from the database and storing it in temporary tables while already making new time-based variables.

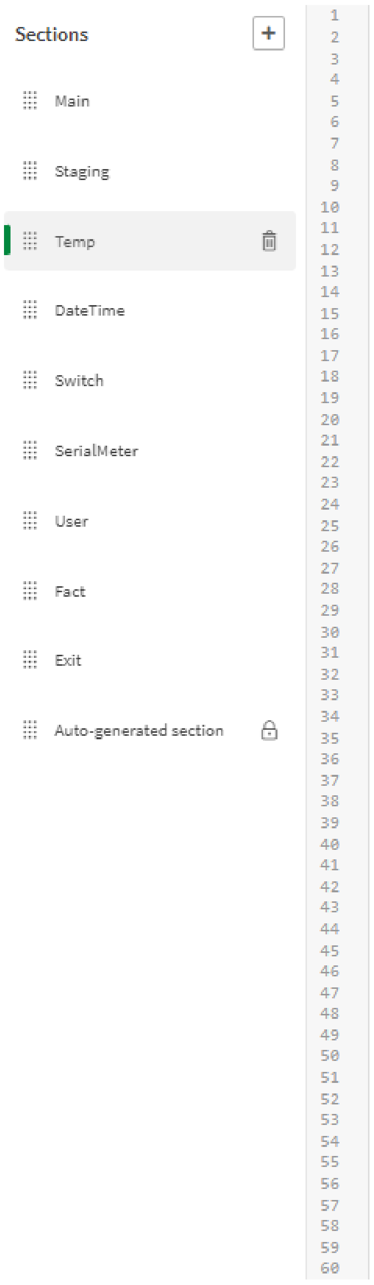


```
LIB CONNECT TO 'test:EnergyApplication';

LIB CONNECT TO 'EnergyApplication';

LOAD id,
    Left(tijdstip, Len(Trim(tijdstip)) - 15) as tijdstip,
    time(Left(tijdstip, Len(Trim(tijdstip)) - 7)) as Time,
    Hour(Left(tijdstip, Len(Trim(tijdstip)) - 7)) as Hour,
    Minute(Left(tijdstip, Len(Trim(tijdstip)) - 7)) as Minutes,
    switch,
    serialMeter,
    currentRate,
    totalDayConsumption,
    totalNightConsumption,
    totalDayConsumption+totalNightConsumption as totalConsumption,
    totalDayProduction,
    totalNightProduction,
    totalDayProduction+totalNightProduction as totalProduction,
    l1Consumption,
    l2Consumption,
    l3Consumption,
    currentConsumption,
    l1Production,
    l2Production,
    l3Production,
    currentProduction,
    l1Voltage,
    l2Voltage,
    l3Voltage,
    l1Current,
    l2Current,
    l3Current;

[wimh_electricity]:
SELECT id,
    tijdstip,
    switch,
    serialMeter,
    currentRate,
    totalDayConsumption,
    totalNightConsumption,
    totalDayProduction,
    totalNightProduction,
    l1Consumption,
    l2Consumption,
    l3Consumption,
    currentConsumption,
    l1Production,
    l2Production,
    l3Production,
    currentProduction,
    l1Voltage,
    l2Voltage,
    l3Voltage,
    l1Current,
    l2Current,
    l3Current
FROM EnergyApplication.dbo."wimh_electricity";
```



```
LOAD id as user_id,
    password,
    last_login,
    is_superuser,
    username,
    first_name,
    last_name,
    email,
    is_staff,
    is_active,
    date_joined;

[auth_user]:
SELECT id,
    password,
    "last_login",
    "is_superuser",
    username,
    "first_name",
    "last_name",
    email,
    "is_staff",
    "is_active",
    "date_joined"
FROM EnergyApplication.dbo."auth_user";


LOAD id as owner_id,
    user_id;

[users_profile]:
SELECT id,
    "user_id"
FROM EnergyApplication.dbo."users_profile";

LOAD id as serialMeter_ID,
    serialNumber as serialMeter,
    owner_id,
    name;

[users_serialnumber]:
SELECT id,
    serialNumber,
    "owner_id",
    name
FROM EnergyApplication.dbo."users_serialnumber";
```

In the next section we start by using the previously made variables to make new ones for our extra intervals while also combining the user-tables into 1 so that their data is synchronized and will be easier to separate later into the correct tables.



```
Temp:
NoConcatenate
LOAD
    tijdstip,
    Time,
    Hour,
    Minutes,
    timestamp(
    IF(Hour = 4 AND Minutes = 0, tijdstip,
    IF(Hour = 8 AND Minutes = 0, tijdstip,
    IF(Hour = 12 AND Minutes = 0, tijdstip,
    IF(Hour = 16 AND Minutes = 0, tijdstip,
    IF(Hour = 20 AND Minutes = 0, tijdstip,
    IF(Hour = 0 AND Minutes = 0, tijdstip, null()))))))) as Every4htemp,
    timestamp(IF(Hour = 23 AND Minutes = 45, tijdstip, null())) as Every24h,
    Date(IF(Hour = 23 AND Minutes = 45, tijdstip, null())) as Datetemp,
    switch,
    serialMeter,
    currentRate,
    totalDayConsumption,
    totalNightConsumption,
    totalConsumption,
    totalDayProduction,
    totalNightProduction,
    totalProduction,
    l1Consumption,
    l2Consumption,
    l3Consumption,
    currentConsumption,
    l1Production,
    l2Production,
    l3Production,
    currentProduction,
    l1Voltage,
    l2Voltage,
    l3Voltage,
    l1Current,
    l2Current,
    l3Current
Resident [wimh_electricity];

NoConcatenate
Users:
LOAD owner_id,
    user_id
Resident [users_profile];

left join (Users)
LOAD user_id,
    username
Resident [auth_user];

left join (Users)
LOAD serialMeter_ID,
    serialMeter,
    owner_id,
    name
Resident [users_serialnumber];
```

Our next few sections are split up in the different dimensions that we can use in our dashboards. Most of them are connected to the Fact-table except our user dimension which is connected to the Serial Meter dimension. This is because of the relation we use in our data frame.

```
DateTime:
NoConcatenate
LOAD AutoNumber(tijdstip, 1) as DateTime_ID,
tijdstip,
Time,
Hour,
Minutes,
Date("Datetemp",'DD/MM') as Date,
Date("Time",'hh:mm') as Every15min,
Date("Every4htemp",'DD/MM hh:mm') as Every4h,
Every24h
Resident Temp;
```

```
Switch:
NoConcatenate
LOAD AutoNumber(switch, 2) as Switch_ID,
switch
Resident Temp;
```

```
SerialMeter:
NoConcatenate
LOAD AutoNumber(serialMeter, 3) as serialMeter_ID,
    AutoNumber(username, 4) as user_ID,
    serialMeter,
    owner_id,
    name
Resident Users;
```

```
User:
NoConcatenate
LOAD Distinct AutoNumber(username, 4) as user_ID,
username
Resident Users;
```

Our Fact-table on the other hand supplies us with the measures that we are going to use. These are the variables which we use to make calculations or to show exact values extracted from the smart meter.

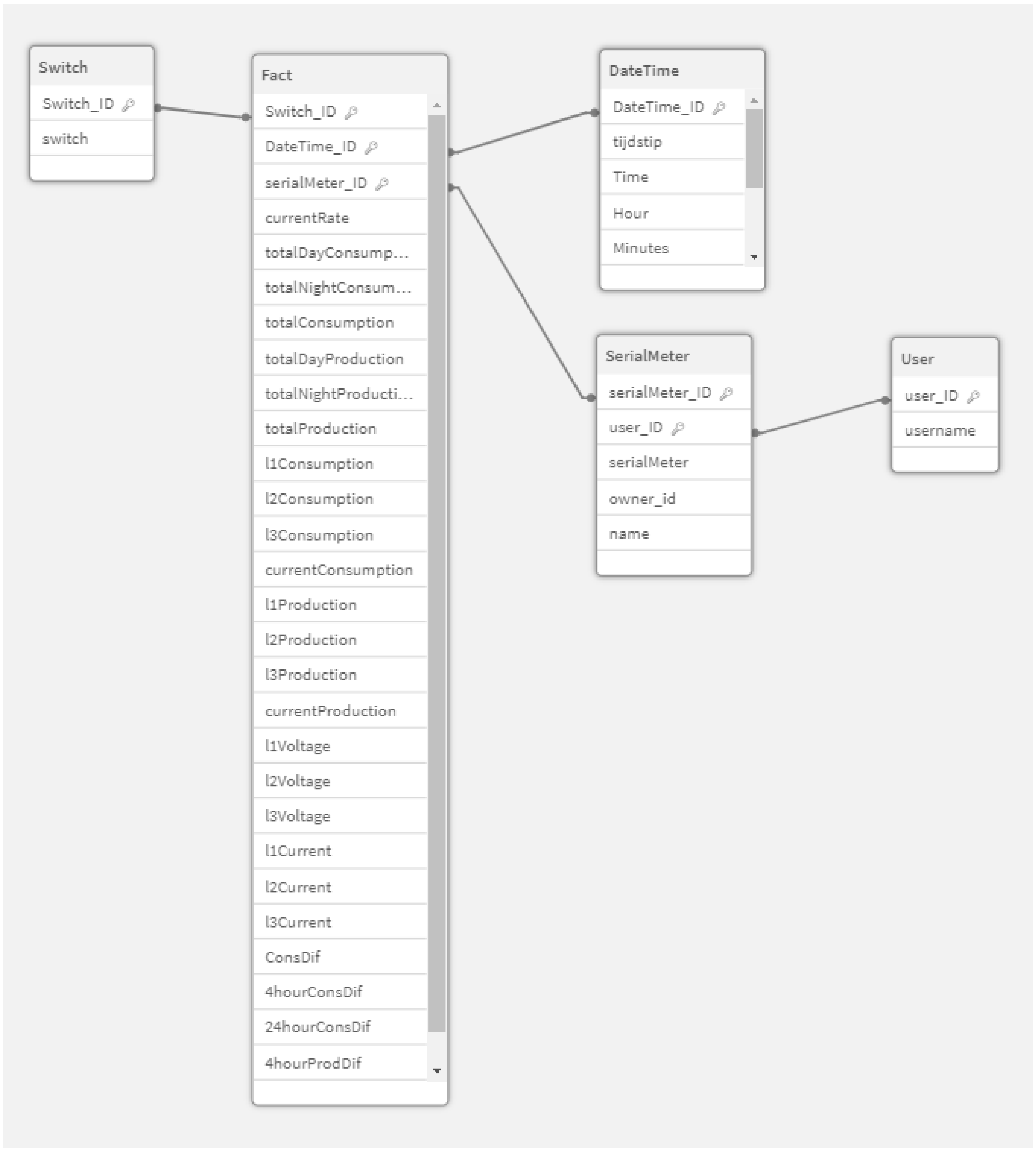
```
Fact:
NoConcatenate
LOAD
    AutoNumber(tijdstip, 1) as DateTime_ID,
    AutoNumber(switch, 2) as Switch_ID,
    AutoNumber(serialMeter, 3) as serialMeter_ID,
    currentRate,
    totalDayConsumption,
    totalNightConsumption,
    totalConsumption,
    if(isnull(Time), null(),num(totalConsumption - Peek(totalConsumption, -1, Time), '##,##')) as "ConsDif",
    if(isnull(Every4htemp), null(),num(totalConsumption - Peek(totalConsumption, -16, Time), '##,##')) as "4hourConsDif",
    if(isnull(Every24h), null(),num(totalConsumption - Peek(totalConsumption, -96, Time), '##,##')) as "24hourConsDif",
    totalDayProduction,
    totalNightProduction,
    totalProduction,
    if(isnull(Every4htemp), null(),num(totalProduction - Peek(totalProduction, -16, Time), '##,##')) as "4hourProdDif",
    if(isnull(Every24h), null(),num(totalProduction - Peek(totalProduction, -96, Time), '##,##')) as "24hourProdDif",
    l1Consumption,
    l2Consumption,
    l3Consumption,
    currentConsumption,
    l1Production,
    l2Production,
    l3Production,
    currentProduction,
    l1Voltage,
    l2Voltage,
    l3Voltage,
    l1Current,
    l2Current,
    l3Current
Resident Temp;


```

And of course, we must close our temporary tables to ensure that they do not interfere with our data frame.

```
Drop Table [wimh_electricity];
Drop Table [auth_user];
Drop Table [users_serialnumber];
Drop Table [users_profile];
Drop Table Temp;
Drop Table Users;
```

Data Frame

Our data frame is simple. We have our Fact-table which is the centre of our data. To our Fact-table we have connected all our other data which we use as dimensions or filters on the data in the Fact-table. There is 1 exception to this which is the User-table. This is because a User can have multiple Serial Meters (which are the unique identifiers for the Smart Meters), and a Serial Meter has more lines of data connected to 1 Serial Meter. This requires a different relation between our tables and results in what is called a snowflake schema.

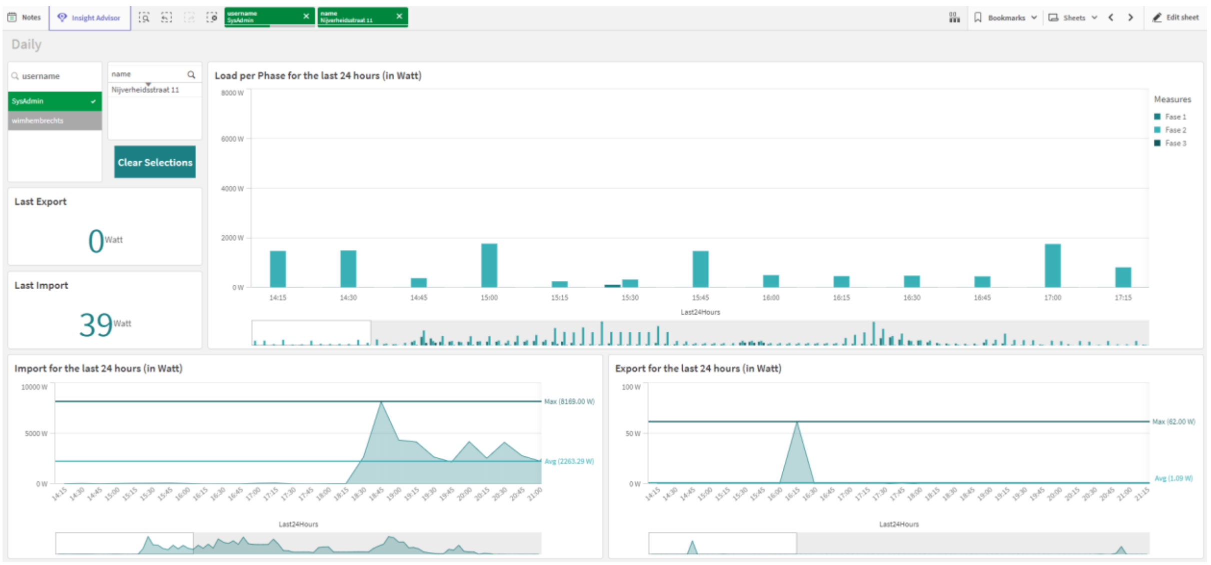


Qlik sense dashboard

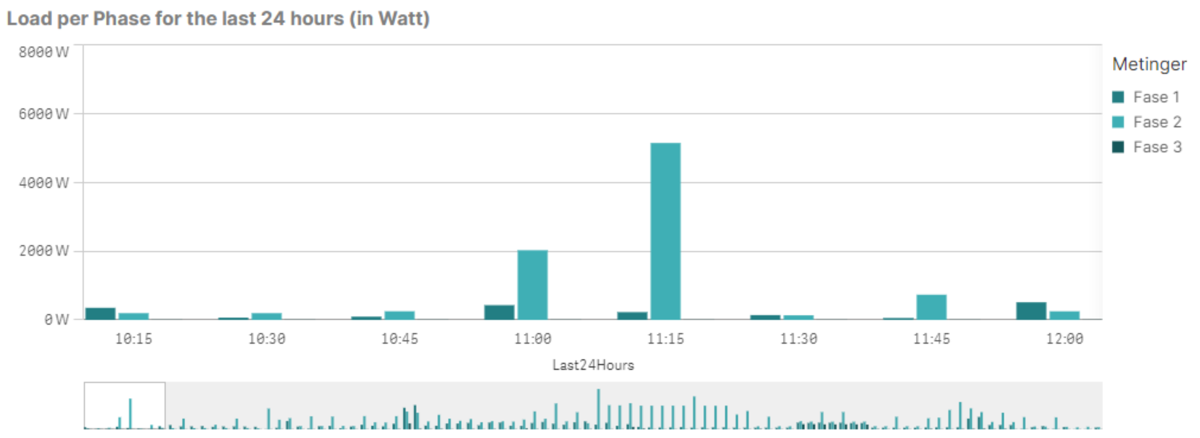
We wanted to make it possible for people to look back in time for about a month which we think is good. So, we made 3 different dashboards, 1 for an entire day with an interval of 15 minutes, another for an entire week with an interval of 4 hours, and finally 1 for an entire month with an interval of 1 day. Sadly enough we did not manage to make it possible to change last 24 hours to the last 24 hours of a given time. We were able to show the import and export of the data to and from the network. But also show the separate phases that are used.

Something which is also quite useful is that 1 person can have multiple meters connected to 1 account. We also made sure that only 1 meter can be selected at once because otherwise there would be a lot of confusion. Plus, data will not be visible if a meter is not selected.

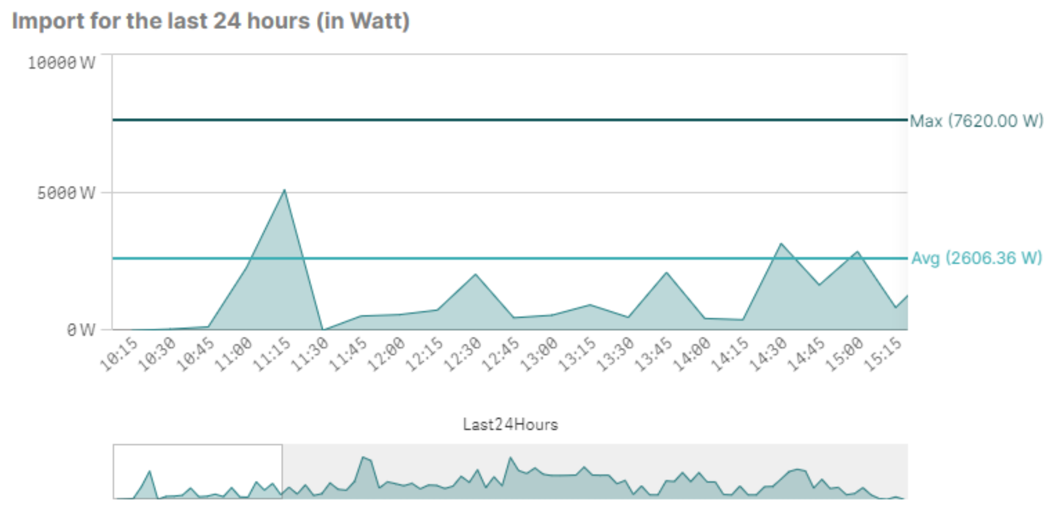
This is the dashboard for the last 24 hours. The other 2 look the same but the data is a little different because of the intervals.



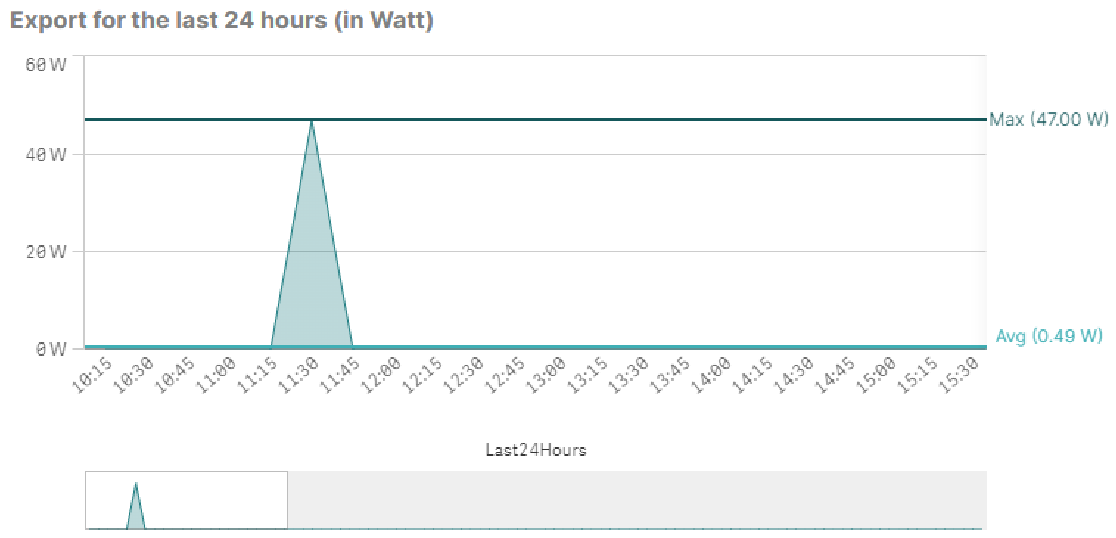
The first graph shows the load per phase for the last 24 hours. How much is each phase being used with segments of 15 minutes



The second graph shows the import in Watt. This explains how much electricity you use at certain time segments divided by 15 minutes. You can also see the max and the average Watt from the current data that is shown.



The third graph shows the export in Watt. This explains how much electricity you generate at certain time segments divided by 15 minutes. You see the max and the average Watt from the current data that is shown as well as the previous graph.

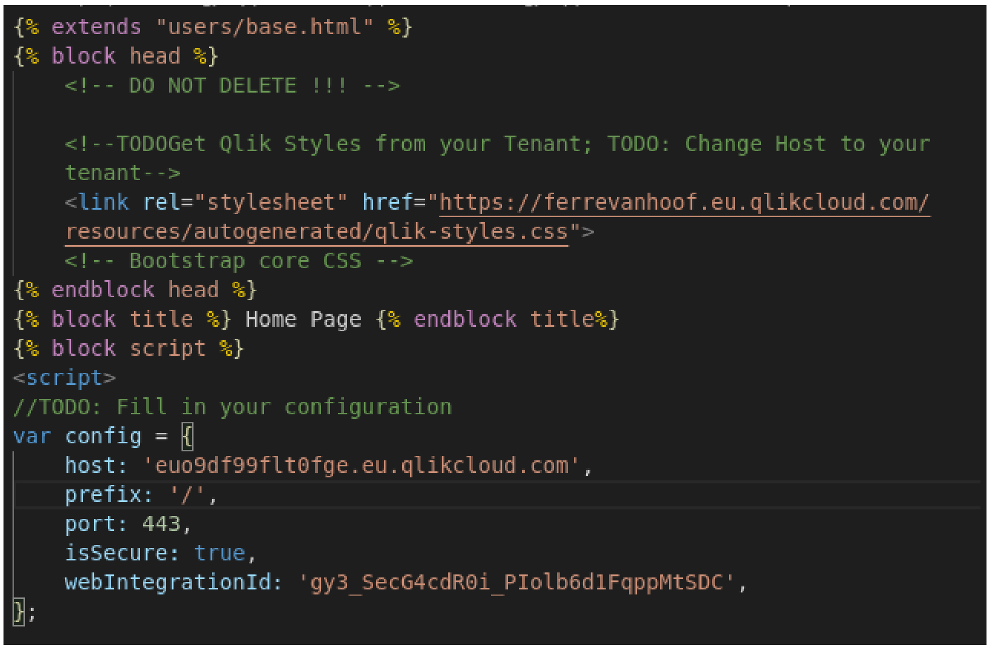


Web Integration

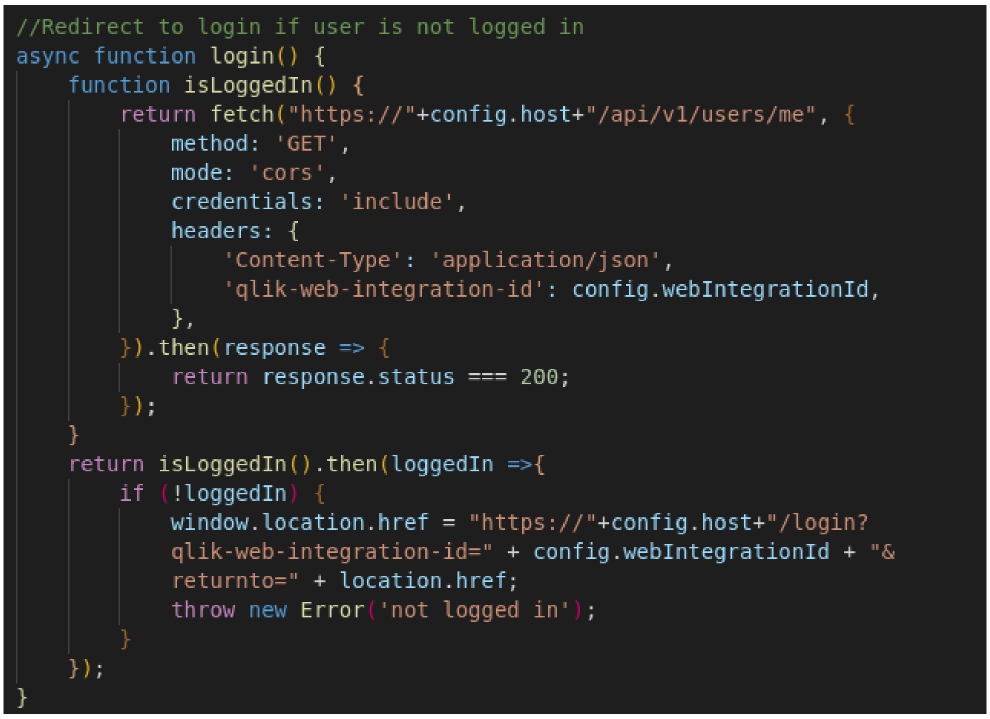
In our code we first make a connection to the tenant that is hosting the application. Afterwards we ask for a login which is required for the type of Qlik account that we use. When this is done, we can link the app we want to show and get the IDs from the specific visualizations that we want to show on our web-application.

To be able to make sure that the logged user sees their own data we made a pre-selection upon login. The selections are not shown on the web-app itself because all users would be able to change those for other people and we do not want that.

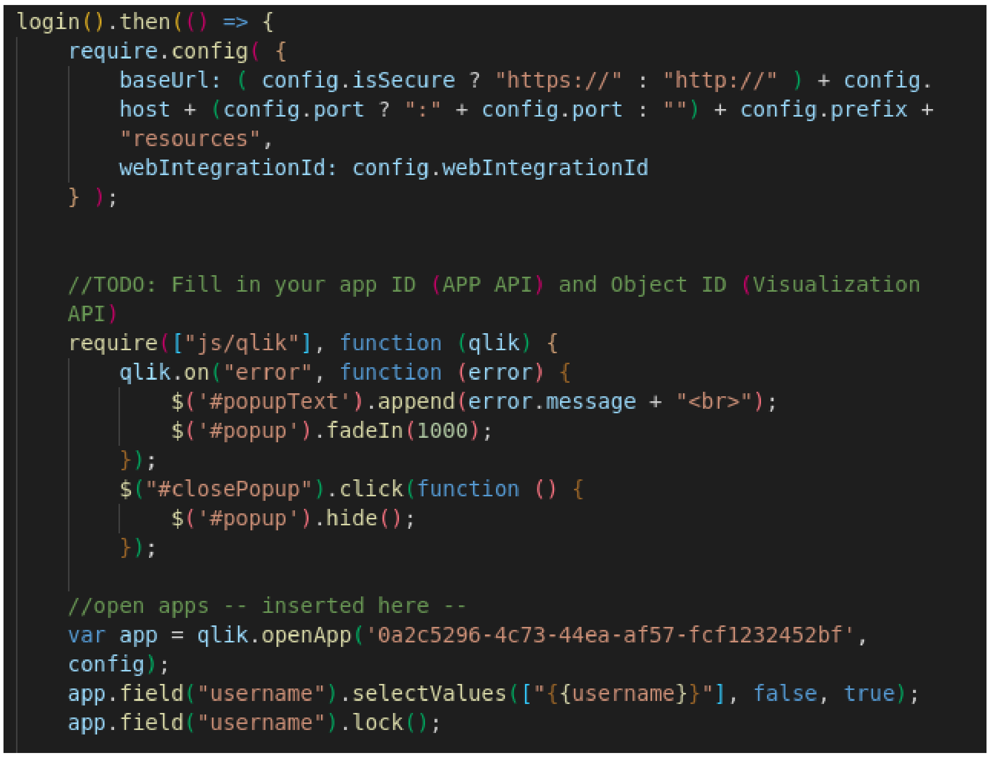
Another slight problem that we have is that because we have a business account and not an enterprise account. The visualizations that we show are all connected to 1 Qlik-app. Which means that its best practice for only 1 user to use the application at the same time. This would not be the case for an enterprise account, but this is way above our budget.



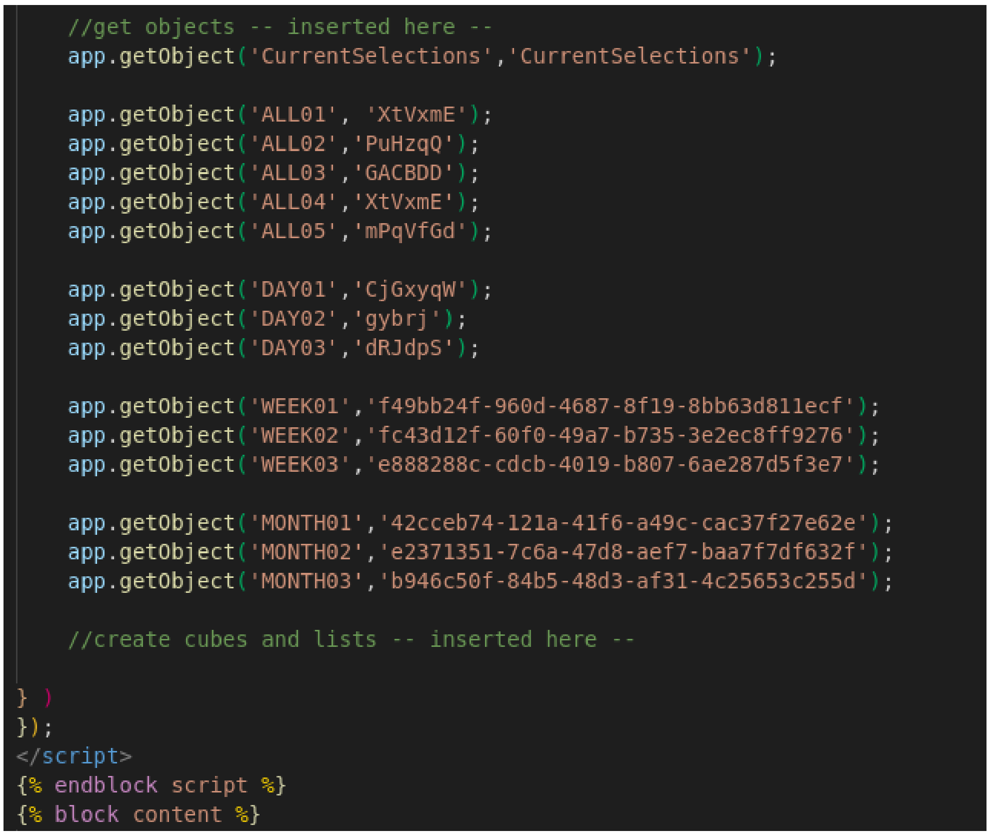
```
{% extends "users/base.html" %}
{% block head %}
    <!-- DO NOT DELETE !!! -->

    <!--TODOGet Qlik Styles from your Tenant; TODO: Change Host to your
    tenant-->
    <link rel="stylesheet" href="https://ferrevanhoof.eu.qlikcloud.com/
    resources/autogenerated/qlik-styles.css">
    <!-- Bootstrap core CSS -->
{% endblock head %}
{% block title %} Home Page {% endblock title%}
{% block script %}
<script>
//TODO: Fill in your configuration
var config = {
    host: 'euo9df99flt0fge.eu.qlikcloud.com',
    prefix: '/',
    port: 443,
    isSecure: true,
    webIntegrationId: 'gy3_SecG4cdR0i_PIolb6d1FqppMtSDC',
};
```



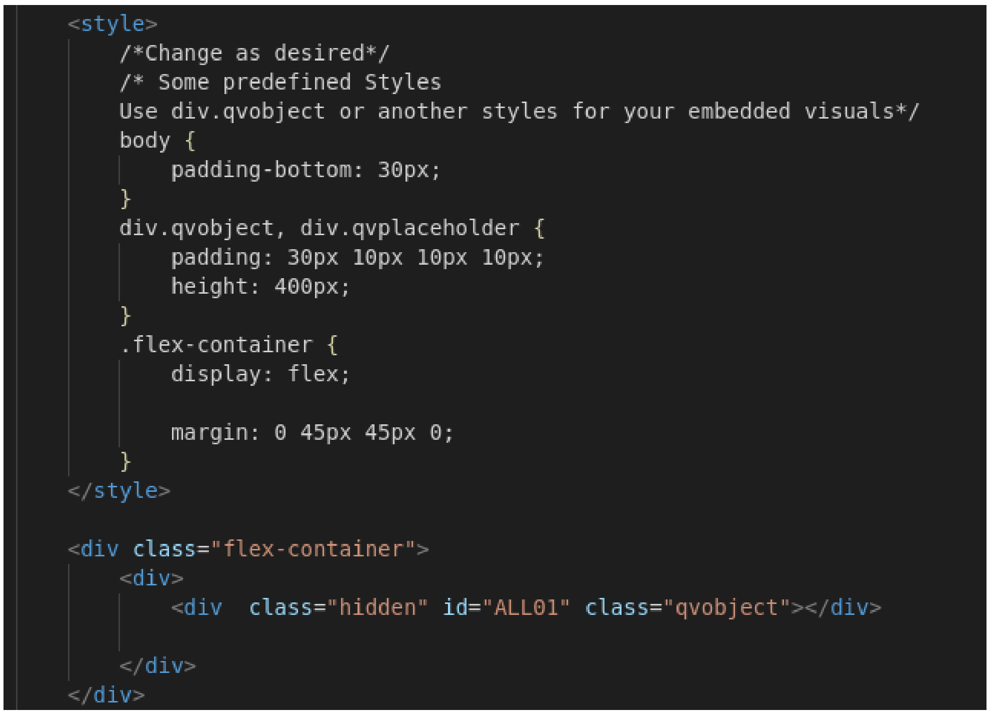
```
//Redirect to login if user is not logged in
async function login() {
    function isLoggedIn() {
        return fetch("https://"+config.host+"/api/v1/users/me", {
            method: 'GET',
            mode: 'cors',
            credentials: 'include',
            headers: {
                'Content-Type': 'application/json',
                'qlik-web-integration-id': config.webIntegrationId,
            },
        }).then(response => {
            return response.status === 200;
        });
    }
    return isLoggedIn().then(loggedIn =>{
        if (!loggedIn) {
            window.location.href = "https://"+config.host+"/login?
            qlik-web-integration-id=" + config.webIntegrationId + "&
            returnto=" + location.href;
            throw new Error('not logged in');
        }
    });
}
```



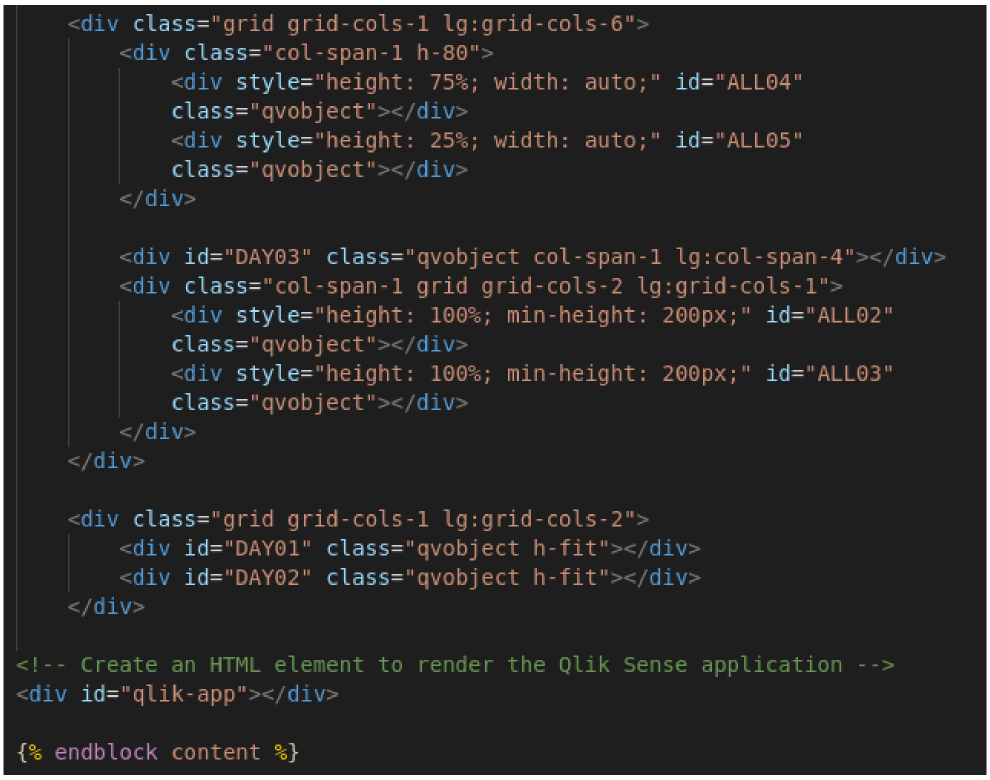
```
login().then(() => {
    require.config( {
        baseUrl: ( config.isSecure ? "https://" : "http://" ) + config.
        host + (config.port ? ":" + config.port : "") + config.prefix +
        "resources",
        webIntegrationId: config.webIntegrationId
    } );


    //TODO: Fill in your app ID (APP API) and Object ID (Visualization
    API)
    require(["js/qlik"], function (qlik) {
        qlik.on("error", function (error) {
            $('#popupText').append(error.message + "<br>");
            $('#popup').fadeIn(1000);
        });
        $("#closePopup").click(function () {
            $('#popup').hide();
        });

    //open apps -- inserted here --
    var app = qlik.openApp('0a2c5296-4c73-44ea-af57-fcf1232452bf',
    config);
    app.field("username").selectValues(["{{username}}"], false, true);
    app.field("username").lock();
```



```
    //get objects -- inserted here --
    app.getObject('CurrentSelections','CurrentSelections');

    app.getObject('ALL01', 'XtVxmE');
    app.getObject('ALL02','PuHzqQ');
    app.getObject('ALL03','GACBDD');
    app.getObject('ALL04','XtVxmE');
    app.getObject('ALL05','mPqVfGd');

    app.getObject('DAY01','CjGxyqW');
    app.getObject('DAY02','gybrj');
    app.getObject('DAY03','dRJdpS');

    app.getObject('WEEK01','f49bb24f-960d-4687-8f19-8bb63d811ecf');
    app.getObject('WEEK02','fc43d12f-60f0-49a7-b735-3e2ec8ff9276');
    app.getObject('WEEK03','e888288c-cdcb-4019-b807-6ae287d5f3e7');

    app.getObject('MONTH01','42cceb74-121a-41f6-a49c-cac37f27e62e');
    app.getObject('MONTH02','e2371351-7c6a-47d8-aef7-baa7f7df632f');
    app.getObject('MONTH03','b946c50f-84b5-48d3-af31-4c25653c255d');

    //create cubes and lists -- inserted here --

} )
});
</script>
{% endblock script %}
{% block content %}
```



```
    <style>
        /*Change as desired*/
        /* Some predefined Styles
        Use div.qvobject or another styles for your embedded visuals*/
        body {
            padding-bottom: 30px;
        }
        div.qvobject, div.qvplaceholder {
            padding: 30px 10px 10px 10px;
            height: 400px;
        }
        .flex-container {
            display: flex;

            margin: 0 45px 45px 0;
        }
    </style>

    <div class="flex-container">
        <div>
            <div  class="hidden" id="ALL01" class="qvobject"></div>

        </div>
    </div>
```



```html
    <div class="grid grid-cols-1 lg:grid-cols-6">
        <div class="col-span-1 h-80">
            <div style="height: 75%; width: auto;" id="ALL04"
            class="qvobject"></div>
            <div style="height: 25%; width: auto;" id="ALL05"
            class="qvobject"></div>
        </div>

        <div id="DAY03" class="qvobject col-span-1 lg:col-span-4"></div>
        <div class="col-span-1 grid grid-cols-2 lg:grid-cols-1">
            <div style="height: 100%; min-height: 200px;" id="ALL02"
            class="qvobject"></div>
            <div style="height: 100%; min-height: 200px;" id="ALL03"
            class="qvobject"></div>
        </div>
    </div>

    <div class="grid grid-cols-1 lg:grid-cols-2">
        <div id="DAY01" class="qvobject h-fit"></div>
        <div id="DAY02" class="qvobject h-fit"></div>
    </div>

<!-- Create an HTML element to render the Qlik Sense application -->
<div id="qlik-app"></div>

{% endblock content %}
```