СОДЕРЖАНИЕ

ВВЕДЕНИЕ 4
1 Теоретические основы разработки *web*-приложений на *Python* 6
1.1 Основы взаимодействия в сети 6
1.2 Работа с *API web*-приложений 8
1.3 Организация хранения данных 9
1.4 Основы построения приложений на базе фреймворка *Django* 11
1.5 Обработка данных на сервере 13
1.6 Обзор дополнительных инструментов 15
2 Реализация *web*-приложения для управления умным домом 16
2.1 Постановка задачи 16
2.2 Обзор архитектуры решения 17
2.3 Изменение данных с помощью экранной формы 19
2.4 Фоновые проверки контроллеров умного дома и принятие решений ... 20
ЗАКЛЮЧЕНИЕ 21
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 22
ПРИЛОЖЕНИЕ А Документы по результатам прохождения курса 23
ПРИЛОЖЕНИЕ Б Листинг программы 24

ВВЕДЕНИЕ

Разработка многофункциональных *web*-приложений на сегодняшний день является одним из самых массовых направлений в сфере информационных технологий. Важнейшим фактором массового распространения информационных технологий явилась возможность их удаленного развертывания, позволяющая осуществлять подключение многих клиентов, имеющих разное географическое расположение и отличающиеся аппаратные средства.

Исторически первые *web*-приложения не выделялись особой функциональностью – это были статические *web*-сайты, при реализации которых разработчиком не закладывалось какой-либо внутренней бизнес-логики и взаимодействия с другими такими же сайтами. Со временем базовые технологии, применяемые в *web*-разработке, такие как язык гипертекстовой разметки *HTML*, таблицы стилей *CSS*, язык браузерного программирования *JavaScript* расширялись новыми стандартами [1]. Так *HTML* и *CSS* последовательно развивали свои возможности по адаптации *web*-контента для различных устройств пользователей, а *JavaScript* обзавелся большим количеством функциональных возможностей, которые поставили его в один ряд с самыми востребованными языками программирования, такими как *Python*.

С другой стороны, по мере поэтапного развития технологий и повсеместного распространения *web*-сервисов появился тренд на *web*-фреймворки – связанные единой архитектурой наборы многофункциональных библиотек для работы с отдельными модулями *web*-приложений в рамках одного языка программирования. Фреймворк *Django*, разработанный для языка *Python*, быстро набрал популярность за счет активного роста сообщества разработчиков этого языка в последние годы.

Он, как и любой полноценный *web*-фреймворк, включает в себя функционал для развертывания локального web-сервера, предназначенного для разработки, имеет собственный модуль для абстрагированного от работы с *SQL* запросами взаимодействия с базами данных *Django ORM*, позволяет используя функциональный подход к программированию реализовывать логику маршрутизации в приложении и включает в себя базовые возможности для валидации и обработки форм с контролируемом жизненным циклом (предполагает объектно-ориентированный подход).

В рамках выполнения курсового проекта предполагается изучения курса «Создание *Web*-сервисов на *Python*» на платформе для онлайн-обучения "Coursera. Авторы курса подробно рассматривают теоретические основы разработки современных *web*-технологий, а также знакомят с широким стеком инструментов, акцентируя внимание на фреймворке *Django*, архитектуре приложений, построенных на нем, и особенностей, связанных с обработкой и хранением данных

Цель курсового проекта: приобрести практические навыки создания интернет-приложений на языке *Python* для повышения уровня компетенций разработчика полного цикла.

Для достижения поставленной цели нужно решить следующие задачи:

1) изучить теоретические основы сетевого взаимодействия;
2) научиться работать с внешними *API*;
3) рассмотреть современные способы хранения данных;
4) изучить основы фреймворка *Django*;
5) научиться получать и обрабатывать данные на сервере;
6) рассмотреть дополнительные инструменты, применяемые в *web*-разработке;
7) используя полученные теоретические знания реализовать полноценное *web*-приложение с использованием возможностей фреймворка *Django* на языке *Python*.

1 Теоретические основы разработки *web*-приложений на *Python*

В процессе прохождения первых шести недель предложенного курса рассматриваются теоретические вопросы, связанные с сетевым взаимодействием и созданием *web*-приложений на языке программирования *Python* с использованием фреймворка *Django*.

1.1 Основы взаимодействия в сети

Компьютерные сети представляют собой набор устройств, которые могут обмениваться информацией друг с другом. Обмен информации в сети должен соответствовать следующим основным требованиям:

– надежность;
– безопасность;
– масштабируемость;
– скорость.

Для теоретического описания сетевого взаимодействия в настоящее время используются модели *OSI* и *TCP/IP* [2]. Структурные схемы рассматриваемых моделей представлены на рисунке 1.

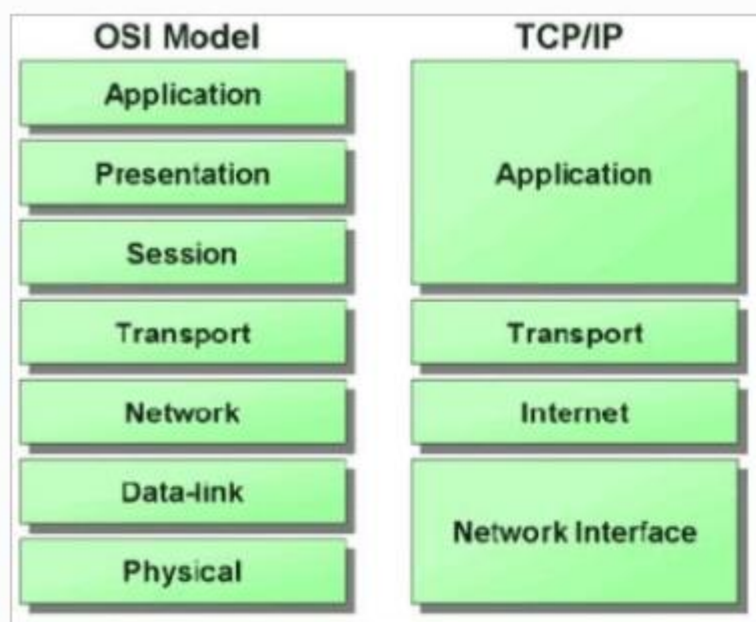

Рисунок 1 – Модели *OSI* и *TCP/IP*

В основе описания взаимодействия устройств лежат инкапсуляция и деинкапсуляция данных. Обе модели предусматривают стек из нескольких уровней. Инкапсуляция предполагает, что при отправке сообщения в сети на прикладном уровне (*Application*) оно проходит через весь стек сверху вниз, и

на каждом уровне к нему добавляется специфический для текущего уровня заголовок. При этом, заголовки, добавленные на предыдущих уровнях, сохраняются. Деинкапсуляция предполагает обратный процесс и имеет место в процессе получения сообщений в сети.

В настоящее время широкое распространение получила модель *TCP/IP*. В основе любого *web*-приложения лежит идея обмена сообщений между сервером и клиентами по протоколам, установленным сетевой моделью. Самым часто используемым протоколом для обмена информации между приложениями является протокол прикладного уровня модели *TCP/IP* – *HTTP*.

Для выполнения *HTTP* запросов в языке программирования *Python* существует библиотека *requests*. Для передачи содержательных данных в сети интернет принято использовать структурированные форматы данных, наиболее применяемым из которых является *JSON* [3].

Ниже представлен пример выполнения *GET* запроса на языке *Python* с использованием библиотеки *requests*. Запросы этого типа используются для получения клиентом актуальных данных с сервера.

```
import requests
r = requests.get('http://httpbin.org/get')
print(r.text)
```

Результатом выполнения запроса является следующий объект, представленный в формате *JSON*:

```
{
  "args": {},
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.18.4",
    "X-Amzn-Trace-Id": "Root=1-5fe405bb-4b46a2bd48dd6e4b1a37cb8c"
  },
  "origin": "217.151.22.163",
  "url": "http://httpbin.org/get"
}
```

Ниже представлен пример выполнения *POST* запроса на языке *Python* с использованием библиотеки *requests*. Запросы этого типа используются для передачи клиентских данных на сервер с целью их обновления или обработки.

```
r = requests.post('http://httpbin.org/post')
print(r.text)
```

POST запрос аналогично *GET* возвращает в качестве ответа объект в формате *JSON*.

Существуют и другие типы *HTTP* запросов, такие как *PUT*, *DELETE* и другие. Они дополняют функциональность рассмотренных запросов и используются в архитектуре *REST API*.

1.2 Работа с *API web*-приложений

С развитием сетевых технологий стало очевидным, что помимо взаимодействия непосредственных участников компьютерных сетей (клиенты и серверы) на уровне протоколов необходимо связать между собой сами *web*-приложения на программном уровне. В основе взаимодействия приложений в сети лежит использование прикладных программных интерфейсов – *API*.

Использование интерфейсов, позволяющих программам обмениваться данными, привело к развитию архитектурного стиля *REST*, предполагающего обмен *HTTP* запросами между клиентским и серверным приложениями. Запрос к данным посредством обращения к *API* сервера позволяет избежать последующего парсинга результата и получить информацию в пригодном для применения виде. Необходимо отметить, что далеко не все *API* являются публичными – большинство программных интерфейсов требуют авторизованного взаимодействия. В этом случае помимо параметров запроса как правило передается некоторый ключ, идентифицирующий пользователя *API*.

Ниже представлен код, позволяющий с помощью запроса *GET* получить детальную информацию о пользователе социальной сети «ВКонтакте» посредством передачи его уникального *ID* в качестве параметра запроса. Дополнительными параметрами могут передаваться названия полей возвращаемого *JSON* объекта, которые требуется получить.

```
import requests
resp = requests.get(
'https://api.vk.com/method/users.get',
params={ 'user_id': '210700286',
#'v': '5.68',
#'fields': 'photo_id, verified, sex, bdate, city, country, home_town,
has_photo, photo_50, photo_100, photo_200_orig, photo_200,
photo_400_orig, photo_max, photo_max_orig, online, domain, has_mobile,
contacts, site, education, universities, schools, status, last_seen,
followers_count, occupation, nickname, relatives, relation, personal,
connections, exports, wall_comments, activities, interests, music,
movies, tv, books, games, about, quotes, can_post, can_see_all_posts,
can_see_audio, can_write_private_message, can_send_friend_request,
is_favorite, is_hidden_from_feed, timezone, screen_name, maiden_name,
crop_photo, is_friend, friend_status, career, military, blacklisted,
```

Таблица 1 – Назначение основных файлов приложения на *Django*

Имя файла	Уровень декомпозиции	Назначение
manage.py	Проект	Запуск приложения на локальном сервере или в режиме *production* с подключением необходимой функциональности фреймворка (команда *python manage.py runserver* выполняет запуск сервера для разработки на локальном компьютере)
settings.py	Проект	Установка константных значений основных параметров приложения. В частности, константными как правило является адрес *API*, к которому обращается приложение, и соответствующий ключ доступа (для работы с *API* в авторизованном режиме)
views.py	Приложение	Обработка запросов, поступающих на сервер, посредством реализации функций, возвращаемыми значениями которых являются ответы сервера на запросы клиента. Помимо этого, данный файл обычно содержит логику валидации и последующей обработки форм

Продолжение таблицы 1

Имя файла	Уровень декомпозиции	Назначение
models.py	Приложение	Описание моделей данных, для их последующего сохранения в базе данных посредством *Django ORM* – технологии фреймворка, позволяющей абстрагироваться от прямых запросов *SQL* и организующей взаимодействие с базой данных с помощью объектно-ориентированного подхода
urls.py	Проект	Маршрутизация проекта – с помощью данного файла описывается логика переходов по адресам для рендеринга требуемых компонентов, входящих в различные приложения в рамках текущего проекта

Функциональность фреймворка позволяет реализовывать приложения практически любой сложности. Более подробно, на примере, использование описанных подходов и файлов будет продемонстрировано на примере создания *web*-приложения для управления умным домом.

1.5 Обработка данных на сервере

Как было сказано выше, для обработки приходящих на сервер данных и подготовке к отправке информации клиентам, предназначен файл *views.py*. Внутри него с использованием различных подходов (объектно-ориентированные формы или обработка запросов на основе методов) происходит обработка информации главным образом с использованием объекта типа *HttpRequest*.

6) отправка изменений на хостинг *Heroku* с помощью *Git* для последующей удаленной сборки и развертывания;

7) повторять шаги 3 – 7 (реже 2 – 7) в случае изменения требуемой логики.

2 Реализация *web*-приложения для управления умным домом

В рамках прохождения курса предлагается выполнить практическое задание, заключающееся в создании приложения для управления умным домом на основе фреймворка *Django*.

2.1 Постановка задачи

Пусть существует некоторый умный дом, представляемый совокупностью датчиков (контроллеров). Дом, как целостная система, обладает собственным *API*, позволяющем получать и изменять состояния одних контроллеров в зависимости от текущего состояния других.

Необходимо реализовать сервер управления умным домом, включающий:

– *web*-интерфейс для настройки и ручного управления;

– модуль, производящий периодический опрос датчиков и осуществляющий автоматическую реакцию в случае возникновения определенных ситуаций.

Опрос контроллеров должен происходить раз в пять секунд в фоновом режиме. *Web*-форма должна открываться в корне сайта и иметь четыре элемента управления умным домом. Целевые значения температуры воды и спальни должны сохраняться в базе данных и использоваться при периодическом опросе контроллеров.

Выделяются следующие особые ситуации, на которые необходимо реагировать в фоновом режиме:

1) в случае, если обнаружена протечка воды (*leak_detector==true*), необходимо закрыть холодную (*cold_water=false*) и горячую (*hot_water=false*) воду и отослать письмо в момент обнаружения;

2) если холодная вода (*cold_water*) закрыта, немедленно выключить бойлер (*boiler*) и стиральную машину (*washing_machine*) и ни при каких условиях не включать их, пока холодная вода не будет снова открыта;

3) если горячая вода имеет температуру (*boiler_temperature*) меньше, чем *hot_water_target_temperature* - 10%, нужно включить бойлер (*boiler*), и ждать пока она не достигнет температуры *hot_water_target_temperature* + 10%, после чего в целях экономии энергии бойлер нужно отключить;

4) если шторы частично открыты (*curtains* == «*slightly_open*»), то они находятся на ручном управлении – это значит, что их состояние нельзя изменять автоматически ни при каких условиях;

5) если на улице (*outdoor_light*) темнее 50, открыть шторы (*curtains*), но только если не горит лампа в спальне (*bedroom_light*). Если на улице (*outdoor_light*) светлее 50, или горит свет в спальне (*bedroom_light*), закрыть шторы. Кроме случаев, когда они на ручном управлении;

6) если обнаружен дым (*smoke_detector*), немедленно выключить следующие приборы [*air_conditioner*, *bedroom_light*, *bathroom_light*, *boiler, washing_machine*], и ни при каких условиях не включать их, пока дым не исчезнет;

7) если температура в спальне (*bedroom_temperature*) поднялась выше *bedroom_target_temperature* + 10% - включить кондиционер (*air_conditioner*), и ждать пока температура не опустится ниже *bedroom_target_temperature* - 10%, после чего кондиционер отключить.

2.2 Обзор архитектуры решения

Разработанное приложение построено на стандартной для фреймворка *Django* файловой схеме.

Запуск локального сервера осуществляется после установки необходимых пакетов и их зависимостей в виртуальном окружении с помощью инструмента *pipenv* (необходимые пакеты описаны в файле *Pipfile*).

Фоновый опрос контроллеров реализован с помощью библиотеки *celery* – раз в пять секунд (настраивается в файле *celery.py*) запускается задача, описанная файле *tasks.py*, выполняющая опрос контроллеров и принятие решений.

Экранная форма реализована с использованием объектно-ориентированного подхода. Класс формы описан в файле *forms.py*, а шаблон – в файле *control.html*.

Авторами курса описана модель сущности для хранения в базе данных и взаимодействия с соответствующей таблицей с помощью *Django ORM*. База данных является встраиваемой (*SQLite*3) и физически расположена в папке с сервером – файл *db.sqlite*3.

Файл *settings.py* дополнен константными полями для хранения адреса *API*, к которому обращается приложение, ключа доступа к нему, а также значения, связанные с отправкой электронных писем.

Необходимо отметить, что согласно принципам построения приложений на *Django* в текущей реализации имеется проект *SmartHouse* и вложенное в

него приложение *coursera_house*. С учетом того, что проект состоит из единственного приложения, эти понятия можно отождествлять.

На рисунке 3 представлена файловая структура разработанного приложения.

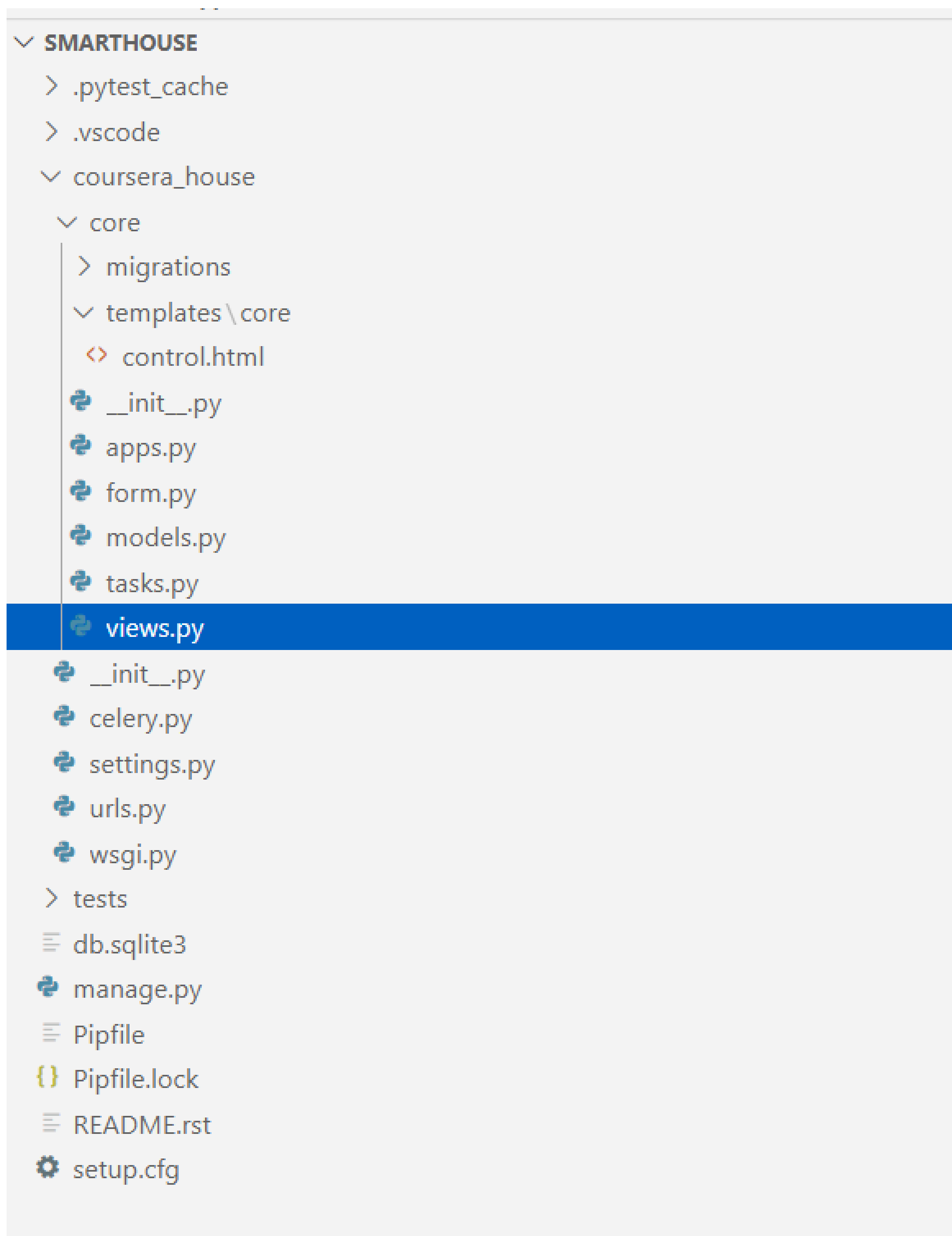

Рисунок 3 – Файловая структура разработанного приложения

2.3 Изменение данных с помощью экранной формы

В существующей реализации экранная форма для управления умным домом расширяет функциональность класса *FormView* и переопределяет его методы жизненного цикла.

Так, переопределенный метод *get* исполняет метод *get_context_data* в том случае, если *API* доступен, иначе возвращает статус ошибки 502 (*Bad Gateway*).

Форма инициализируется начальными значениями в переопределенном методе *get_initial*.

При отправке формы, в случае корректного ввода данных, автоматически вызывается переопределенный метод *form_valid*, выполняющий сохранение целевых температур в базе данных и изменяющий состояние контроллеров.

Метод *create_or_update_db_entry* пробует получить запись по ключу – по имени контроллера. В случае, если запись в БД не найдена – в таблицу добавляется новая запись. Иначе существующая запись обновляется и сохраняется.

Ниже представлено содержимое файла *form.py*, в котором описан класс экранной формы, включающий типы данных и правила валидации.

```python
from django import forms
from django.core.validators import MinValueValidator, MaxValueValidator

class ControllerForm(forms.Form):
    bedroom_target_temperature = forms.IntegerField(
        required=False,
        validators=[MinValueValidator(16), MaxValueValidator(50)]
    )
    hot_water_target_temperature = forms.IntegerField(
        required=False,
        validators=[MinValueValidator(24), MaxValueValidator(90)]
    )
    bedroom_light = forms.BooleanField(required=False)
    bathroom_light = forms.BooleanField(required=False)
```

Полный исходный код разработанного приложения представлен в Приложении Б.

На рисунке 4 представлен скриншот экранной формы в браузере.

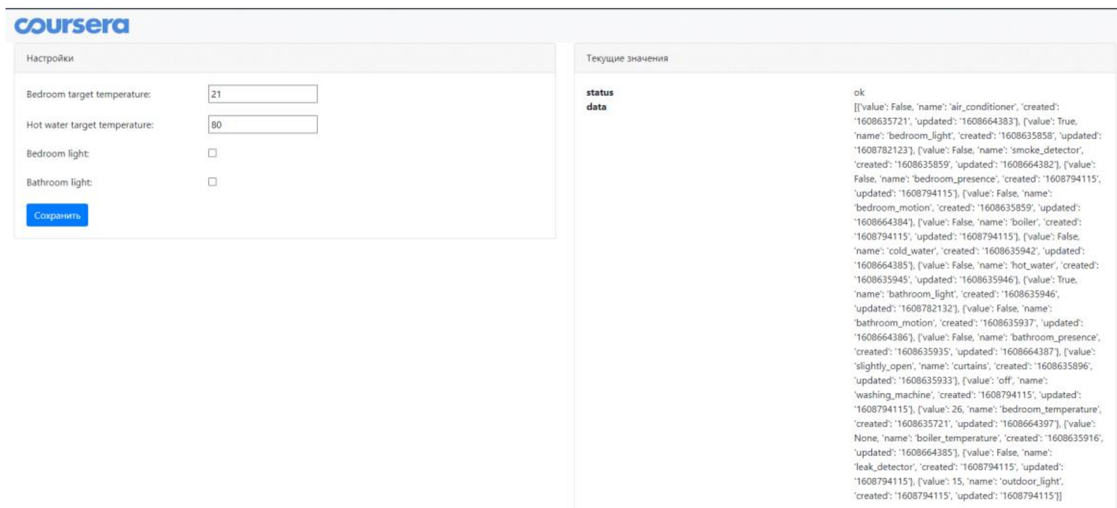

Рисунок 4 – Скриншот экранной формы в браузере

2.4 Фоновые проверки контроллеров умного дома и принятие решений

Задача (*task*) выполняющая опрос контроллеров в фоновом режиме каждые пять секунд описана в файле *tasks.py* в методе *smart_house_manager*.

На первом этапе выполняется *GET* запрос на получение состояний каждого из контроллеров, после чего инициализируется пустой словарь *new_states*, в который будут помещаться сведения о контроллерах, состояние которых должно измениться (ключом будет являться имя контроллера, значением – новое состояние).

Далее, на основании описанных в задании ситуаций заполняем словарь *new_states*. При проверке уделяем внимание ситуации, когда контроллер, который необходимо выключить, уже находится в выключенном состоянии – в этом случае ему не требуется менять состояние. В том случае, если после всех проверок в словаре отсутствуют какие-либо записи, запроса на сервер не отправляется.

Иначе отправляем *POST* запрос, куда, согласно документации к *API*, предоставленной авторами курса, передаем *JSON* объект с нужной структурой. Таким образом, в случае, если состояние каких-либо контроллеров изменилось, каждые пять секунд к *API* будут отправляться два запроса. Если состояние менять не требуется – один запрос.

Система оценивания практических заданий *Coursera* учитывает количество выполненных запросов. Оптимизация количества запросов в единицу времени (максимальное количество обычно ограничивается поставщиком серверных услуг) является ключевым фактором быстродействия *web*-приложений.

ЗАКЛЮЧЕНИЕ

В процессе выполнения курсового проекта изучены теоретические основы сетевого взаимодействия между *web*-сервисами, рассмотрена технология разработки *web*-приложений на языке программирования *Python* с использованием фреймворка *Django*.

Изучены вопросы, связанные с долговременным хранением данных, валидацией и обработкой клиентских форм, обработкой *HTTP* запросов, работой с *API* приложений, созданных сторонними разработчиками.

Завершен курс «Создание *Web*-сервисов на *Python*» на образовательной платформе *Coursera*. В процессе прохождения предложенного курса изучены востребованные инструменты разработки и развертывания приложений; практически отработаны теоретические материалы при выполнении заданий по программированию с автоматической проверкой.

Результатом выполненной работы является реализация *web*-приложения для управления умным домом, включающая в себя работу с клиентской формой, регулярное обновление данных и авторизированное взаимодействие с внешним *API*.

В результате выполнения курсового проекта получены важные теоретические сведения об особенностях *web*-разработки на *Python*, а также общих подходах, применяемых при создании современных *web*-приложений. На практике закреплены навыки разработки приложений на *Python* с использованием фреймворка *Django*.

ПРИЛОЖЕНИЕ А
(обязательное)

Документы по результатам прохождения курса

На рисунке А.1 представлен результат автоматического оценивания разработанного *web*-приложения системой тестирования *Coursera*.

Гуненков Михаил ...

урсовой проект «Web-приложение для управлени

Пред. | Дальше

Инструкции **Моя работа** Обсуждения

+ Создать работу

Ваши работы

Дата	Результат	Зачет?
23 декабря 2020 г., 18:27 +06	25/25	Да
Приложение для управления умным домом	25/25	Скрыть результаты анализатора

```
Congratulations! All tests passed.
```

Рисунок А.1 – Результат оценивания разработанного *web*-приложения

На рисунке А.2 представлен сертификат, подтверждающий прохождение рассматриваемого курса.

Рисунок А.2 – Сертификат об окончании курса

ПРИЛОЖЕНИЕ Б
(обязательное)

Листинг программы

Файл *manage.py*:

```
#!/usr/bin/env python
import os
import sys

if __name__ == "__main__":
    os.environ.setdefault("DJANGO_SETTINGS_MODULE", "coursera_house.settings"
)
    try:
        from django.core.management import execute_from_command_line
    except ImportError:
        # The above import may fail for some other reason. Ensure that the
        # issue is really that Django is missing to avoid masking other
        # exceptions on Python 2.
        try:
            import django
        except ImportError:
            raise ImportError(
                "Couldn't import Django. Are you sure it's installed and "
                "available on your PYTHONPATH environment variable? Did you "
                "forget to activate a virtual environment?"
            )
        raise
    execute_from_command_line(sys.argv)
```

Файл *celery.py*:

```
from __future__ import absolute_import, unicode_literals
import os
import django
from celery import Celery

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'coursera_house.settings')
django.setup()

app = Celery('proj')
app.config_from_object('django.conf:settings', namespace='CELERY')
app.autodiscover_tasks()

from coursera_house.core.tasks import smart_home_manager

@app.on_after_configure.connect
def setup_periodic_tasks(sender, **kwargs):
    sender.add_periodic_task(5, smart_home_manager.s(), name='Check Smart Hom
e')
```

Файл *settings.py*:

```
import os
from celery.schedules import crontab
# Build paths inside the project like this: os.path.join(BASE_DIR, ...)
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))

# Quick-start development settings - unsuitable for production
# See https://docs.djangoproject.com/en/1.11/howto/deployment/checklist/

# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = 'dv1qcb=38l1xxc46xpp#qi(hz548)2o+2#z*0@xof=dkvl+ahk'

# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = True

ALLOWED_HOSTS = []

# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'coursera_house.core'
]

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]

ROOT_URLCONF = 'coursera_house.urls'

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': ["templates"],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
```