Создание Web-сервисов на Python

Московский физико-технический институт и Mail.Ru Group

Неделя 6

Содержание

1. Telegram бот

1.1. Что такое Telegram бот

Чат-боты позволяют нам создавать текстовые интерфейсы для взаимодействия с пользователями и поддерживается большим количеством платформ для обмена сообщениями. Удобство чат-ботов перед веб-сайтами и мобильные приложения для разработчиков в том, что ему не нужно задумываться о том, как делать дизайн или как поддерживать большое количество устройств. Все это осуществляется платформой, на которой запускается чат-бот. Также сейчас чат-боты позволяют обмениваться не только текстовыми сообщениями, но и передавать фото, видео, встраивать локации и обмениваться платежами. Таким образом вы можете легко запустить сервис, доступный десяткам миллионов пользователей с привычным для них интерфейсом.

1.2. Регистрация и создание простого бота

- Регистрация бота происходит в чате с @BotFather
- Команда /help позволяет вывести список доступных команд
- Для создания бота — команда /newbot
- В результате вы получите токен вида 473874797:AAFvdrsM4-9w7m0LIHbJrzemvgK2ZH3KWOY

Нужно написать некий набор функций, который нужно вызывать при прихое сообщения от пользователя. Для этого поможет готовое рещение, бибиотека telebot.

```
import telebot
token = '473874797:AAFvdrsM4-9w7m0LIHbJrzemvgK2ZH3KWOY'
```

Передаем токен.

```
bot = telebot.TeleBot(token)
```

Бот создан, но еще ничего не делает. Нужно его запустить. Мы это сделаем с помощью polling – переодической проверки Telegram сервера на предмет новых сообщений. Для того, чтобы бот обрабатывал сообщения, полученные от пользователя, нужен handle_message. Здесь наш бот просто распечатает текст сообщения.

```
    def handle_message(message):
        print(message.text)
    bot.polling()
```

Теперь мы хотим связать функцию с реальным действием. Нам нужно написать декоратор, который есть в библиотеке telebot.

```
    @bot.message_handler()
    def handle_message(message):
        print(message.text)
    bot.polling()
```

Связали эту функцию с отправкой сообщения. Теперь мы хотим, чтобы бот что-то писал в ответ. Воспользуемся функцией send_message и укажем, что посылать и куда.

```
    @bot.message_handler()
    def handle_message(message):
        print(message.text)
        bot.send_message(chat_id=message.chat.id,text='Узнай курс валют.')
    bot.polling()
```

1.3. Расширенная обработка сообщений

Хотим сделать бота, который сообщал бы нам курс валют. Для этого нам интересно узнать, какую валюту спрашивает пользователь.
Простой пример – он, допустим, хочет узнать курс доллара и пишет нам сообщение ”курс доллара”. Мы можем понять его желания путем проверки наличия в его сообщении одной из списка валют, которые мы знаем. Например, возьмём две валюты – доллар и евро, и мы хотим определить, есть ли упоминание хоть одной из них в нашем сообщении. Можем в нашем декоратор message handler указать аргумент – функцию, которую он будет проверять, соответствует ли сообщение, которое мы сейчас получили, какие-то паттернам, каким-то условиям, и если функция возвращает ”да”, то этот обработчик подходит, и вывод выполнен. А если он вернет ”нет”, то не подходит, и мы пойдём к следующему обработчику.
Давайте напишем обработчик, который будет определять, есть ли одна из списка валют в нашем списке пользователя. Сначала произведем список валют. Возьмём две валюты- евро и доллар. Отлично. И напишем функцию, которая принимает на вход сообщение и возвращает true или false, в зависимости от того, есть эта валюта в списке или нет. Мы назовем ее check_currency, она принимает сообщение, и смотрим. Идём по нашему списку валют for c in currencies и если эта валюта есть в тексте сообщения, а текст сообщения у нас содержится в атрибуте текста message.txt.

Мы возвращаем true и если мы прошли весь цикл и не нашли ни одной из подходящих валют, то возвращаем false.

```
currencies = ['евро', 'доллар']

def check_currency(message):
    for c in currencies:
        if c in message.text.lower():
            return True
    return False
```

И теперь мы напишем ещё один message handler, который будет вызываться тогда, когда эта валюта найдена в сообщении. В наш декоратор мы укажем, какая функция должна проверять подходимость этого сообщения. Это делается при помощи аргумента func и потом check_currency. И теперь давайте просто выведем какое-нибудь сообщение – не узнаем курс валют, если там была валюта, а "ты спросил про валюту".

```
currencies = ['евро', 'доллар']

def check_currency(message):
    for c in currencies:
        if c in message.text.lower():
            return True
    return False
```

Запустим нашего бота и посмотрим что получится. Пишем доллар. Он пишет "ты спросил про валюту". Пишем "привет", он пишет "узнать курс валют".
Осталось найти подходящий курс валют. Мы сделаем это, просто используя словарь курсов валют. Воспользуемся функцией check_currency_value которое принимает на вход текст сообщения и пытается найти, что там была за валюта, и из словаря значений возвращает одно из значений. Если она не нашла значения, то она не возвращает ничего. Будем теперь использовать эту функцию в нашем разработчике сообщений. Пишем check_currency_value и передаем туда текст сообщения message.txt. Оно выдает нам пару значений currency и value. Если никакая валюта не была найдена, напишем то же самое – "узнать курс валют".

```
def check_currency_value(text):
    currency_values = {'евро': 70, 'доллар': 60}
    for currency, value in currency_values.items():
        if currency in text.lower():
            return currency, value
    return None, None

@bot.message_handler(commands=['rate'])
@bot.message_handler(func=check_currency)
def handle_currency(message):
    print(message)
    currency, value = check_currency_value(message.text)
    if currency:
        bot.send_message(message.chat.id, text='Курс {} равен {}.'.format(currency,
        ↪  value))
    else:
        bot.send_message(message.chat.id, text='Узнай курс валют.')
```

Мы можем помочь пользователю, воспользовавшись командами. Команды – это общий принцип для многих платформ, где сообщения начинаются с /, название команды, и потом вы пишете что-то после этого. И таким образом, пользователь может увидеть, какие команды есть у нашего бота и более явно сообщить своё желание. Для определения того, что нам пришла команда, мы можем в декораторе указать список команд, которые мы хотим обрабатывать этим сообщением. Мы добавили ещё один декоратор поверх нашего обработчика. И теперь указали не func, а commands, в котором мы подали список названий тех команд, в ответ на которые мы хотим указывать курс валют. Назвали эту команду rate.

1.4. Обработка сообщений с разным типом контента

Напишем еще один обработчик для выдачи локации, например, ближайшего банка.

```
def get_closest_bank(location):
    lat = location.latitude
    lon = location.longitude
    bank_address = 'Красноармейская, 20'
    bank_lat, bank_lon = 55.800389, 37.543710
    return bank_address, bank_lat, bank_lon
```

Теперь создадим функцию, которая будет нам эту локацию выдавать.

```
@bot.message_handler(content_types=['location'])
def handle_location(message):
    location = message.location
    bank_address, bank_lat, bank_lon = get_closest_bank(location)
    logo = open('bank.jpg', 'rb')
    bot.send_photo(message.chat.id, logo, caption='Ближайший банк находится по адресу
    ↪ {}'.format(bank_address))
    bot.send_location(message.chat.id, bank_lat, bank_lon)
```

1.5. Кнопки и меню

В Telegram есть такой механизм, как меню. Мы можем создавать некий markup, который позволяет отобразить пользователю набор возможностей или опций, который он может выбрать. Их бывает два: это ReplyKeyboardMarkup и InlineKeyboardMarkup.
ReplyKeyboardMarkup просто заменяет вашу стандартную клавиатуру в телефоне несколькими кнопками, нажав на любую из которых, ваш пользователь может отправить сообщение, к которому вы готовы и в ответ на которое вы отправите определенное сообщение.
InlineKeyboardMarkup — более интересная вещь. Она позволяет вам ваше сообщение, будь то фото или текст, дополнить снизу кнопками, нажав на которые, пользователь может и не отправлять сообщение, а просто взаимодействовать с этим, как с кнопками в мобильном приложении или на сайте. И это позволяет нам делать сложные элементы. Например, вы можете в ответ на нажатие кнопки менять контент сообщения и, например, сделать каталог.
Для того чтобы сделать эту клавиатуру, нам нужно из модуля Telebot импортировать модуль types, в котором есть типы объектов, нужных для создания клавиатуры. Напишем функцию, которая будет принимать список наших валют, которые нам интересны, и создавать клавиатуру с этими названиями, чтобы пользователь знал уже, какие валюты доступны в нашем приложении.
Для начала создадим клавиатуру. Она содержится в модуле types, и называется этот тип объекта InlineKeyboardMarkup. У него есть параметр rows_width, который определяет, сколько вы хотите, чтобы кнопок было в каждой строчке, в каждом ряду. Допустим, если бы у вас было десять валют и вы укажете rows_width = 2, он создаст пять рядов, каждый по две кнопки. Клавиатура есть, теперь нужно добавить в нее кнопки. Давайте создадим их с помощью list comprehension. Эти кнопки обладают двумя свойствами. Первое — это текст, который видит пользователь, а второе — это дата, это строка, которая будет вам передаваться в результате нажатия пользователем на эту кнопку. Мы говорим: for для каждой currency, которая есть в нашем списке валют, создадим новую кнопку types InlineKeyboardButton. И у нее будет текст — собственно сама эта валюта и данные, которые мы передаем, тоже будет название этой валюты. Все очень просто. И сохраним это как список, который называется buttons. И теперь, чтобы добавить эти кнопки в нашей клавиатуре, мы пишем keyboard.add и передаем ей список аргументами создания. Создали клавиатуру, осталось ее вернуть для дальнейшего использования: return keyboard.

```
    def create_keyboard():
        keyboard = types.InlineKeyboardMarkup(row_width=2)
        buttons = [types.InlineKeyboardButton(text=c, callback_data=c)
         for c in currencies]
        keyboard.add(*buttons)
        return keyboard
```

Чтобы ее использовать, нужно нам ее вызвать и получить keyboard. И мы можем передавать ее в любое сообщение как последний аргумент. Помимо текста и чата, куда мы отправляем, мы можем указать, что у нас будет такой reply_markup, то есть наше сообщение будет обладать таким меню, которое мы создали.

```
    @bot.message_handler(commands=['rate'])
    @bot.message_handler(func=currency_in_message)
    def handle_message(message):
        currency, value = check_currency_value(message.text)
        keyboard = create_keyboard()
        if currency:

            bot.send_message(message.chat.id, text='Курс {} равен {}'.format(currency,
             ↪  value),
                             reply_markup=keyboard)
        else:
            bot.send_message(message.chat.id, text='Укажите валюту',reply_markup=keyboard)
```

Мы получили две кнопки: евро и доллар. Нажимаем на любую из них, и ничего не происходит. Давайте обработаем эти нажатия, так же, как если бы пользователь написал: курс евро или курс доллара. То есть проверим, что это за валюта, и напишем ее значение. Для этого нам нужен еще один ряд и нам нужен обработчик. Но обработчик не сообщений, а обработчик вот этого нажатия.
Для этого есть другой декоратор: бот callback_query_handler. Его нужно передать в функцию, в ответ на которую он будет вызываться, в нашем случае всегда. То есть мы прямо здесь напишем: функция — это такая лямбда, которая в ответ на любое сообщение возвращает true. Назовем это callback _handler. И здесь наш параметр — это не message, сообщение, а вот эта query, которая к нам пришла. Назовем ее callback_query. Но у нее есть тоже сообщение, оно находится в атрибуте message. То, что нас интересует, это что, на какую кнопку он нажал. Итого, у нас есть наш текст, есть сообщение, к которому было прикреплено это меню, и мы можем теперь ответить на это нажатие так же, как мы отвечали всегда. Пользуясь нашей функцией check_currency_value, мы передаем туда наш текст и, как и прежде, отправляем сообщение, если есть такая валюта.
Неудобство этого подхода в том, что ваше меню уезжает выше и выше. И возможно, что бы вы хотели сделать, это в ответ на нажатие не отправлять сообщения, а отобразить курс как-то еще. И это уникальный способ, когда вы можете, если ваш бот, например, находится в чате с пользователями и вы не хотите засорять чат большим количеством сообщений, от бота и от пользователей, вы можете сделать набор кнопок, нажав на которые, пользователь, только тот,

который нажал, что-то увидит.
Для этого вам нужно воспользоваться методом не send_message, а answer_callback_query, который покажет сообщение сверху вашего чата. Мы указываем, на какую query мы хотим ответить. На callback_query, которая нам пришла, на ее id. И указываем, что мы хотим ответить. Мы хотим ответить то же самое — каков курс этой валюты. Сверху показывается некая notification. Это очень удобно и позволяет вам делать удобные групповые боты, где множество пользователей в чате общаются и взаимодействуют с неким меню, например, можно сделать голосование, где результат голосования показывается здесь, а не засоряет весь чат.

```
@bot.callback_query_handler(func=lambda x: True)
def callback_handler(callback_query):
    message = callback_query.message
    text = callback_query.data
    currency, value = check_currency_value(text)
    if currency:
        bot.answer_callback_query(callback_query.id, text='Курс {} равен
        ↪  {}'.format(currency, value))
    else:
        bot.send_message(message.chat.id, text='Укажите валюту')
```

2. Git

2.1. Введение в работу с Git

Git - распределенная система управления версиями или DVCS, придуманная Линксом Торвелтсом - создателям linux.
Git – это набор консольных утилит, хотя существуют и графические клиенты, с помощью которых можно контролировать изменения в любых файлов. С git можно откатится на старую версию всего проекта или только отдельных файлов, сравнить версии файлов за разное время, параллельно разрабатывать разные фитчи в проекте и потом объединять эти изменения из разных версий в одну.

Git сохраняет всю информацию об изменениях файлов и о своих настройках в папке.git в корне проекта. Эта папка и называется репозиторий. Работать с git можно локально, а можно синхронизировать свой репозиторий с удаленным репозиторием и это позволяет работать над одним проектом нескольким людям совместно. Есть сервисы для хранения репозиториев, например, сайт github.com, gitlab.com, bitbucket.org.
Попробуем git на практике. Теперь для начала работы нужно настроить два параметра. Это имя пользователя и e-mail. Давайте напишем, для этого мы используем команду git config с параметром – global, что мы настраиваем глобальные настройки не для какого-то конкретного репозитория, а вообще на компьютере. Эти параметры обязательно задавать для того, чтобы у каждого изменения, который вы будете вносить в репозиторий, был правильный адрес. Это

позволяет понимать кто какие правки вносил и нужно для работы.

```
msk-wifi-17fap2-t_abramov-noname:projects t.abramov$ git config --global us
er.name "Timur Abramov"
msk-wifi-17fap2-t_abramov-noname:projects t.abramov$ git config --global us
er.email "abramov@corp.mail.ru"
msk-wifi-17fap2-t_abramov-noname:projects t.abramov$
```

Рис. 1

Давайте теперь создадим папку проекта, назовем ее git_test, перейдем в нее. Теперь мы должны создать в этой папке репозиторий git. Мы пишем git init. После этого мы видим, что появилась папка .git.
Мы находимся в ветке master. Ветки – это такой особый механизм, который позволяет делать несколько версий одних и тех же файлов.
Сейчас у нас одна ветка по умолчанию. Вот мы создали новый репозиторий, теперь давайте создадим файлы проекта. На самом деле репозиторий можно было создать уже в папке с какими-то файлами.
Ну допустим создадим файл hello world, hello.py его назовем и давайте создадим еще какие-то файлы. Ну и давайте создадим ещё какую нибудь директорию, допустим это будет папка logs.

```
msk-wifi-17fap2-t_abramov-noname:git_test t.abramov$ echo "# hello world" >
> hello.py
msk-wifi-17fap2-t_abramov-noname:git_test t.abramov$ touch 11.pyc
msk-wifi-17fap2-t_abramov-noname:git_test t.abramov$ touch 111.pyo
msk-wifi-17fap2-t_abramov-noname:git_test t.abramov$ mkdir logs
msk-wifi-17fap2-t_abramov-noname:git_test t.abramov$ ls -la
total 8
drwxr-xr-x   7 502  staff  224 22 янв 21:45 .
drwxr-xr-x  21 502  staff  672 22 янв 21:44 ..
drwxr-xr-x   9 502  staff  288 22 янв 21:44 .git
-rw-r--r--   1 502  staff    0 22 янв 21:45 11.pyc
-rw-r--r--   1 502  staff    0 22 янв 21:45 111.pyo
-rw-r--r--   1 502  staff   14 22 янв 21:45 hello.py
drwxr-xr-x   2 502  staff   64 22 янв 21:45 logs
msk-wifi-17fap2-t_abramov-noname:git_test t.abramov$
```

Рис. 2

Во-первых, git показывает, что есть несколько неотслеживаемых файлов, но при этом он не показывает вообще папку logs и это правильно. Если в директории нет файлов, то она не участвует в версиаривании, потому что вообще оперирует с файлами. Файлы имеют пути и поэтому могут находиться в директориях, но пустые директории никак не отслеживается.

```
msk-wifi-17fap2-t_abramov-noname:git_test t.abramov$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        11.pyc
        111.pyo
        hello.py

nothing added to commit but untracked files present (use "git add" to track
)
```

Рис. 3

Допустим я хочу еще исключить из отслеживания файлы pyc и pyo. Обычно эти файлы создаются автоматически питоном и содержат скомпилированный код. Они генерируются из файлов pyo и поэтому они не нужны мне в репозитории. Для этого мне нужно записать именно исключаемых файлов и папок в специальный файл.gitignore, причем в нем допускаются маски. Такой файл может находиться в любой папке проекта и влияет на нее и все вложенный в нее папки.

```
msk-wifi-17fap2-t_abramov-noname:git_test t.abramov$ echo "*.py[co]" > .git
ignore
msk-wifi-17fap2-t_abramov-noname:git_test t.abramov$ ls
11.pyc          111.pyo         hello.py        logs
msk-wifi-17fap2-t_abramov-noname:git_test t.abramov$ ls -la
total 16
drwxr-xr-x   8 502  staff  256 22 янв 21:47 .
drwxr-xr-x  21 502  staff  672 22 янв 21:44 ..
drwxr-xr-x   9 502  staff  288 22 янв 21:46 .git
-rw-r--r--   1 502  staff    9 22 янв 21:47 .gitignore
-rw-r--r--   1 502  staff    0 22 янв 21:45 11.pyc
-rw-r--r--   1 502  staff    0 22 янв 21:45 111.pyo
-rw-r--r--   1 502  staff   14 22 янв 21:45 hello.py
drwxr-xr-x   2 502  staff   64 22 янв 21:45 logs
msk-wifi-17fap2-t_abramov-noname:git_test t.abramov$
```

Рис. 4

Мы создали файл gitignore, теперь посмотрим, что нам скажет git status. Файлы pyc и pyo исчезли из сообщения о файлах, которые git видит. В файл gitignore имеет смысл добавлять все папки вроде папок кэшей, папок логов, весь мусор, который не следует хранить вместе с проектом.

А теперь допустим я хочу, чтобы папка logs, которой у нас сейчас не отображается, все таки попала в репозиторий, но файлы, которые я буду добавлять внутри этой папки, не попадали. Мы создадим внутри папки logs файл gitignore, в котором мы укажем игнорировать всё, кроме самого файла gitignore, и это сразу решить две проблемы. Мы будем игнорировать файлы внутри logs и плюс мы поместим хотя бы какой-то файл внутри папки и она у нас появится в git-е.

```
msk-wifi-17fap2-t_abramov-noname:git_test t.abramov$ echo "*" > logs/.gitig
nore
msk-wifi-17fap2-t_abramov-noname:git_test t.abramov$ echo "!.gitignore" >>
logs/.gitignore
-bash: !.gitignore: event not found
msk-wifi-17fap2-t_abramov-noname:git_test t.abramov$ echo '!.gitignore' >>
logs/.gitignore
msk-wifi-17fap2-t_abramov-noname:git_test t.abramov$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore
        hello.py
        logs/

nothing added to commit but untracked files present (use "git add" to track
)
msk-wifi-17fap2-t_abramov-noname:git_test t.abramov$
```

Рис. 5

Git видит теперь все нужные файлы и не видит ненужные, но пока эти файлы не отслеживаемые. Чтобы добавить всё под контроль и подготовить к commit-у, а commit, это есть фиксация текущего состояния файлов, используем команду git add.

```
msk-wifi-17fap2-t_abramov-noname:git_test t.abramov$ git add .
msk-wifi-17fap2-t_abramov-noname:git_test t.abramov$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   .gitignore
        new file:   hello.py
        new file:   logs/.gitignore
```

Рис. 6

У git есть понятие изменений, которые будут включены в ближайший commit, так называемое staging area или область подготовки. Если изменить какой-то файл, то изменения автоматически не попадут в commit, а вот с помощью команды git add мы как раз указываем файлы или целые директории, которые должны попасть в commit.
Теперь давайте зафиксируем состояние, То есть сделаем commit. commit, это что-то вроде снапшота репозиторий на какой-то момент. Чтобы сделать commit, нужно чтобы были хоть какие-то изменения. Для commit требуется осмысленное сообщение, поясняющие какие изменения сделаны.

```
msk-wifi-17fap2-t_abramov-noname:git_test t.abramov$ git commit -m"initial
coomit"
[master (root-commit) daad3a2] initial coomit
 3 files changed, 4 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 hello.py
 create mode 100644 logs/.gitignore
msk-wifi-17fap2-t_abramov-noname:git_test t.abramov$
```

Рис. 7

2.2. Удаленные репозитории и ветки

Теперь давайте внедрим в наш проект новую функциональность. Для этого мы делаем ветку, в которую мы будем вносить изменения. Для этого есть команда got checkout.

```
tabramov:git-test t.abramov$ git checkout -b feature/print-to-console
Switched to a new branch 'feature/print-to-console'
tabramov:git-test t.abramov$ git status
On branch feature/print-to-console
nothing to commit, working tree clean
tabramov:git-test t.abramov$
```

Рис. 8

Назвать ветку можно как угодно, хоть одной буквой, но мы назвали ветку по схеме, где первое слово – это тип изменений, который мы хотим внести, а после слеша у нас идет как можно более краткое описание.
Я открою файл hello и добавлю сюда немножечко print'ов.

```
# hello world
print("hello world")
print("lorem ipsum")
~
```

Рис. 9

Теперь я хочу удалить файл 111.pyc. git status видит изменения только в файле hello.py. Он не видит то, что удален файл 111.pyc, потому что мы его в предыдущем видео специально добавляли в git ignore и git его просто не отслеживает.

```
tabramov:git-test t.abramov$ git status
On branch feature/print-to-console
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   hello.py

no changes added to commit (use "git add" and/or "git commit -a")
tabramov:git-test t.abramov$
```

Рис. 10

Давайте пишем еще одну интересную команду - git diff, она нам покажет какие файлы изменялись и что конкретно в них изменялось с момента последнего коммита.

```
tabramov:git-test t.abramov$ git diff
diff --git a/hello.py b/hello.py
index 2be7c65..f69ba6d 100644
--- a/hello.py
+++ b/hello.py
@@ -1 +1,3 @@
 # hello world
+print("hello world")
+print("lorem ipsum")
```

Рис. 11

Давайте теперь наши изменения добавим с помощью команды git add, как мы это делали ранее и закоммитим с помощью git commit. То есть мы закоммитили новые изменения в нашу ветку. Давайте посмотрим отличия этой ветки от ветки мастер. Теперь у нас git diff ничего не выдаст, потому что мы все изменения закоммитили и теперь изменения из предыдущего комита нет никаких, а вот если мы напишем git diff master, то мы посмотрим отличие текущей ветки от мастер.
А теперь давайте посмотрим еще историю коммитов, можем написать git log и увидеть, что у нас было два коммита и видим их id. Это уникальные номера каждого коммита, по которому можно в принципе переключиться на этот коммит, чекаутнутся в него, можно посмотреть див между двумя комитетами и так далее.

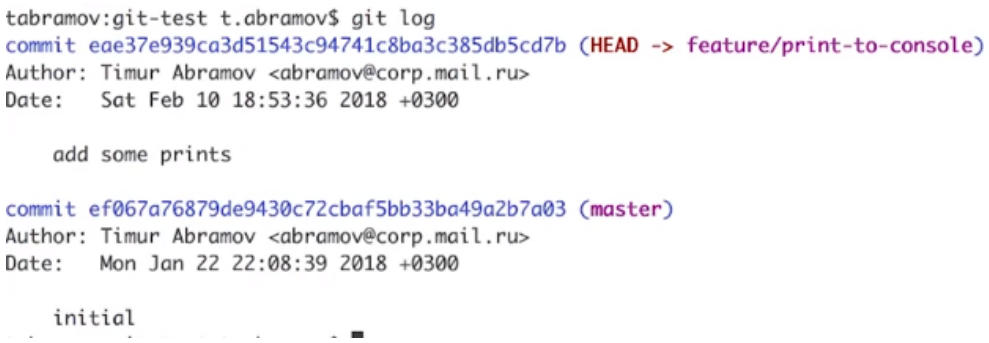

Рис. 12

А теперь давайте переключимься обратно на ветку мастер (git checkout master). Здесь мы не пишем ключ -b, потому что -b – это создать новую ветку.
Все изменения, которые мы внесли в другую ветку, в ветке мастер как бы исчезли.
Теперь давайте отправим все наши изменения в удаленный репозитарий на gitlab.com. На бесплатных тарифных планах можно делать приватный репозиторий, а на гитхабе нельзя.
На git lab есть кнопка new project и создадим новый проект, новый репозиторий, назовем его git-test. Здесь есть инструкции о том, как с ним работать, если в разных ситуациях.
У нас ситуация, когда у нас уже есть локальный гит репозиторий и мы хотим просто добавить удаленный репозиторий к уже существующему. А давайте просто скопируем эту команду и добавим удаленный репозиторий.

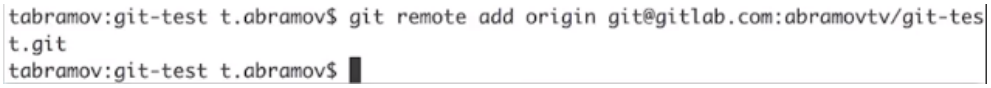

Рис. 13

Если у вас не настроены SSH-ключи, их моно легко настроить, и у вас будут передаваться ваши изменения без спрашивания пароля. Можно выбрать HTTPS, но тогда вам надо будет при отправке изменений ввести логин и пароль. Возможно вам это будет проще сделать, поэтому обратите на это внимание.
Теперь надо отправить изменение всех веток на гитлаб. Для этого есть команда git push -u.

2.3. Merge и конфликты

Нужно создать локальную копию. Достаточно написать git clone, скопировать этот адрес репозитория, который нам выдали, и указать, в какую папку. Если мы не укажем папку, то репозиторий будет скопирован в папку с именем этого репозитория. Мы находимся в ветке master, и никаких других веток локальных в этом репозитории нет. Но если мы напишем git branch −r, мы увидим, что в удаленном репозитории есть ветка print-to-console, которую мы сделали в прошлом.
Давайте доработаем проект и сделаем новую ветку.

```
tabramov:my-project t.abramov$ git branch
* master
tabramov:my-project t.abramov$ git branch -r
  origin/HEAD -> origin/master
  origin/feature/print-to-console
  origin/master
tabramov:my-project t.abramov$ git checkout -b feature/hello-username
Switched to a new branch 'feature/hello-username'
tabramov:my-project t.abramov$
```

Рис. 14

Добавим новый функционал. Есть в sys такой параметр argv, это список аргументов.

```
import sys
who = 'world' if len(sys.argv)<2 else sys.args[1]
print("hello, %s" % who)
```

Рис. 15

Ключ -a к commit автоматически добавляет все измененные файлы в commit. Но если бы мы создавали новые файлы, они бы не были добавлены. То есть это работает только с измененными, поэтому git add в целом надежнее.
Давайте теперь запушим ветку в репозитории на gitlab. Так как мы клонировали наш репозиторий, то родительский репозиторий у нашего локального уже установлен, тот, с которого мы клонировали. Git предлагает запушить нашу локальную ветку в удаленную с таким же именем.

```
tabramov:my-project t.abramov$ git push
fatal: The current branch feature/hello-username has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin feature/hello-username

tabramov:my-project t.abramov$     git push --set-upstream origin feature/hello-use
rname
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 405 bytes | 405.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
```

Рис. 16

Теперь git отправил наши изменения в удаленный репозиторий с тем же именем и связал нашу локальную ветку с удаленной. Вот теперь для отправки изменений из нашей ветки достаточно просто делать git push без каких-то дополнительных параметров.
Смотрите, теперь мы хотим, чтобы ответственный за проект влил наши доработки в ветку

master, и для этого мы поставим ему merge request. Идем на gitlab и ставим. Есть кнопка Create merge request, он как раз по умолчанию последнюю ветку берет и предлагает ее в master. Здесь мы можем указать, кто должен посмотреть наш merge request. Можем поставить галочку Remove source branch, это позволит удалить автоматически ветку, как только она будет вмёрджена в master.
Merge request - это важная часть при коллективной работе. Хотя объединять изменения из разных веток можно и локально с помощью команды git merge и потом отправлять результат в удаленный репозиторий с помощью команды push, но как тогда ревьюить ваш код?
Пусть кто-то решил влить доработки из своей ветки в master без того, чтобы поставить в merge request. Вот этот наш старый репозиторий, смотрим, в какой мы ветке. Мы находимся в ветке feature/print-to-console, и мы просто берем и мёрджим ее в master. До этого мы переключаемся в master и делаем git merge. Спросили наш комментарий для merge, и вот мы объединили. Если вы сливаете ветки фич с мастером, следует указывать ключ —— no—ff. Без него может получиться так, что история коммитов из ветки вольется в master так аккуратно, что будет выглядеть так, будто вы все коммитили в master. А вот с ним будет четко видно, что коммиты были в отдельную ветку, а потом ветка объединилась с master отдельным коммитом.

```
tabramov:git-test t.abramov$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
tabramov:git-test t.abramov$ git merge --no-ff feature/print-to-console
Merge made by the 'recursive' strategy.
 hello.py | 2 ++
 1 file changed, 2 insertions(+)
```

Рис. 17

Теперь кнопка merge стала недоступной, и сбоку написано, что есть merge конфликты. Конфликты появились потому, что в ветке master есть изменения в тех же местах, которые мы хотим изменить, хотели изменить в нашей ветке. Конфликты надо решать вручную или автоматически, несложные можно решить прямо на gitlab, нажав кнопку Resolve conflicts, а более сложные придется решать локально.
Давайте попробуем с этим что-то сделать. Возвращаемся в репозиторий. Давайте сделаем команду git fetch. Она притаскивает изменения, которые произошли в удаленном репозитории, но не объединяет их с локальными ветками, а просто притаскивает. А теперь мы попробуем смёрджить новый мастер в нашу ветку.

```
tabramov:my-project t.abramov$ git fetch
remote: Counting objects: 1, done.
remote: Total 1 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (1/1), done.
From gitlab.com:abramovtv/git-test
   ef067a7..67e6e1f  master     -> origin/master
tabramov:my-project t.abramov$ git merge origin/master
Auto-merging hello.py
CONFLICT (content): Merge conflict in hello.py
Automatic merge failed; fix conflicts and then commit the result.
```

Рис. 18

git merge пишет, что есть конфликты в hello.py. Удалим изменения, пришедшие с сервера, потому что мы считаем, что наши изменения правильные, а те, которые были в master внесены – не очень. И теперь чтобы показать, что все конфликты мы решили, мы пишем git add hello.py, таким образом мы показываем git, что конфликты решены. И можем написать git commit. Нам предлагается сообщение по умолчанию, что мы мёрджим ветку, что были такие-то конфликты, мы их решили, мы согласны с этим. И все, мы можем делать git push, и отправляем нашу обновленную ветку, нашу ветку hello-username, обратно в gitlab.
Теперь смотрим, что произошло. Кнопка merge снова стала доступной, потому что мы притащили последние изменения из master в нашу ветку, решили все конфликты и заново запушили, теперь никаких конфликтов нет, и мы можем смёрджить как надо. Наш merge request подчинился.
Когда вы работаете в команде, лучше изменения все-таки делать в ветке и ставить свой merge request в общем репозитории, чтобы его все посмотрели, указали ошибки, указали, какие ошибки надо устранить. Здесь можно оставлять комментарии к merge requests на gitlab, и на github тоже. И когда все будут довольны, вашу ветку можно будет смёрджить. Сейчас мы это и сделаем.

3. Development и Production

3.1. Отличия Development от Production

Во время разработки на локальном компьютере проект доступен только вам, разработчику, и нагрузки на него нет вообще никакой.
На production ситуация совсем иная. Может прийти сразу много людей и возникнуть большая нагрузка. Могут появится параллельные запросы, и они могут привести к состоянию гонки. Важна безопасность и производительность. Все должно быть заранее скомпилировано, оптимизировано. Статика, такая, как js- и css-файлы, картинки, файлы пользователей, раздаются не вашим приложением, как обычно это бывает в разработке, а специальным оптимизированным под это веб-сервером, например, nginx, или даже сторонним сервисом для доставки контента, CDN. Например, самый известные — это Cloudflare, NetDNA или Amazon CloudFront. Пароль от базы данных, ключи шифрования, данные для авторизации во всевозможных сторонних API должны быть разными в разработке и в production. Во многих компаниях у разработчиков нет доступа к боевым серверам и боевым базам данных, и они не знают пароли и другие боевые реквизиты.
Таким образом, настройки проекта для разработки очень сильно отличаются от настроек этого же проекта при запуске его в production:

- Это касается, например, флагов отладки, которые есть у многих фреймворков, и влияют на вывод стек-трейса на страницу при ошибке и много на что еще на самом деле.
- Это касается секретных реквизитов: паролей, настроек кешей, путей для различных файлов, используемых веб-серверов для запуска приложения и методов раздачи статических файлов, хотя не только этого.

Важной частью является и то, как код разработчиков попадает и запускается на боевых серверах. Нет каких-то общих подходов и шаблонов, и в каждой компании это реализовано по-своему.

Код может запускаться на железном сервере, своем или арендованном, на виртуальной машине, так называемой VPS, в докер-контейнере или на облачном хостинге. Одних только крупных облачных хостингов наберутся десятки. Из известных: Amazon EC2, Microsoft Azure, Google App Engine, Heroku и другие. При этом выбранная вами инфраструктура: VPS, облачный хостинг и так далее, может сильно влиять и на стек используемых вами технологий, и на способы доставки кода из среды разработки на production.
Например, в Heroku достаточно закомить изменения в специальный репозиторий git, для того чтобы запустить код на production. А, например, Google App Engine накладывает ограничения на доступные языки программирования и привязывает проект к своему стеку технологий.
В большинстве компаний также используются системы непрерывной интеграции, обычно на базе чего-то вроде Jenkins или Gitlab CI. Они, при обнаружении изменений в коде в системе контроля версий, автоматически выкачивают эти изменения и запускают сборку проекта и автоматические тесты, проверяют качество кода и его покрытие тестами, а могут заодно и выкатывать код на боевые или тестовые сервера. Это позволяет автоматически тестировать любые изменения, внесенные в код разными членами команды, и обнаруживать проблемы, особенно проблемы интеграции и ошибки на очень ранних этапах. Они также позволяет автоматизировать многие рутинные операции по сборке проекта и подготовке его к production.
Также между разработкой и production существуют еще и stage, или отладочные сервера. Железо и ПО, а также процессы на них, процессы в том числе доставки кода, максимально приближены к production-серверам, но при этом используются они для внутреннего тестирования, то есть недоступны снаружи, но при этом для тестирования в условиях, максимально приближенных к боевым.
Конкретные механизмы: continuous integration, тестирования, релизов на боевые сервера, способы запуска кода в production, мониторинга и уведомления об ошибках, очень сильно отличаются от компании к компании, и даже от проекта к проекту. И разные задачи в этих процессах могут выполняться совершенно различными командами, не только программистами, но и девопсами, тестировщиками, администраторами.

3.2. Раскладка проекта на Heroku

В requirements_txt указаны наши требования — это Django, redis, пакет для связи redis с python нам нужен, поскольку мы храним в redis счетчик просмотров страницы, gunicorn, в production django запускается не через runserver, а через gunicorn или другой веб-сервер, и для запуска на Heroku он также нужен. И модуль django-heroku — это модуль от самого Heroku, он не обязателен, но помогает сконфигурировать настройки Django для работы с Heroku, чем упрощает нам жизнь.

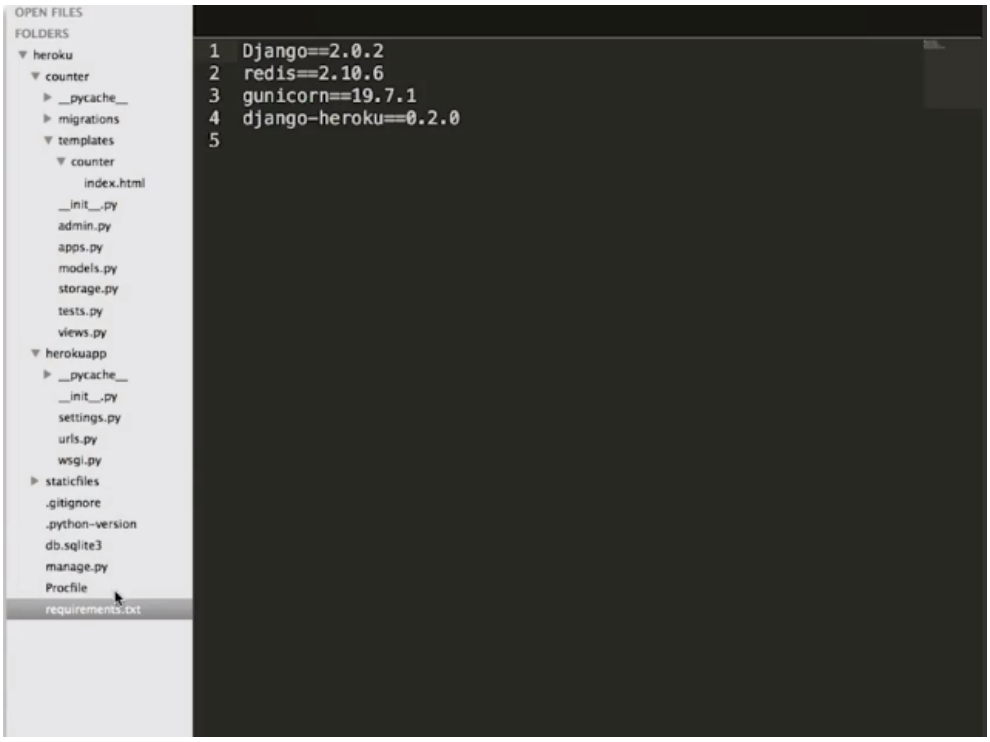

Рис. 19

Также здесь есть файл Procfile, вот этот файл чисто для Heroku. В нем мы указали, как запускать наше веб-приложение. И это минимум, что требуется.

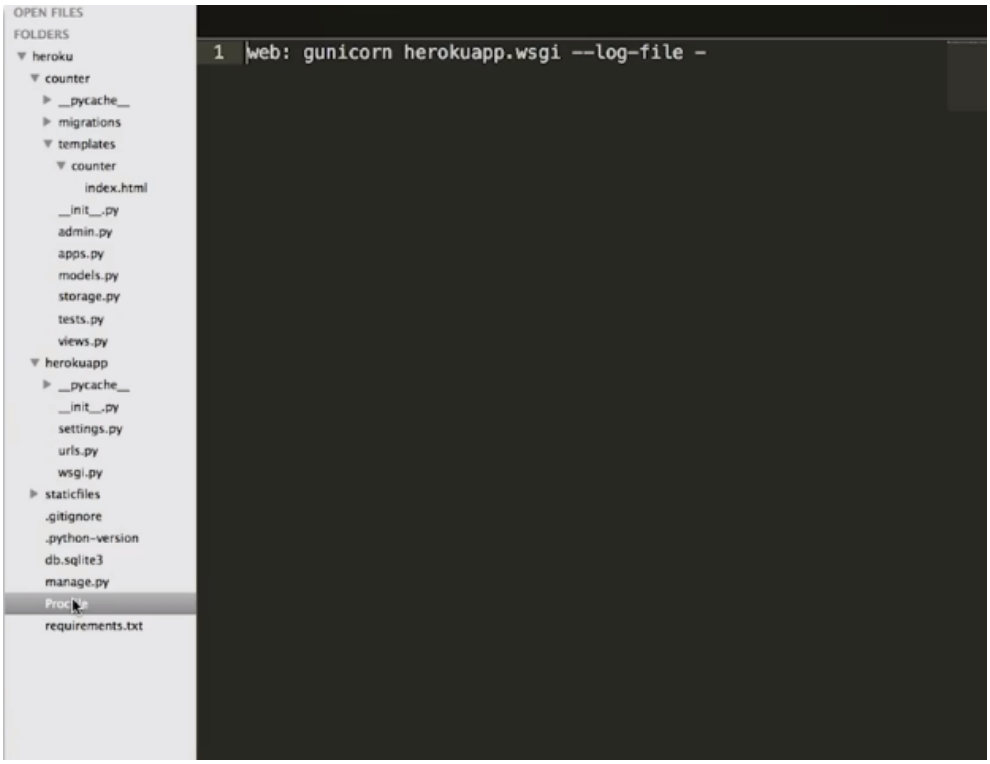

Рис. 20

В этом файле вообще указываются настройки для различных процессов, и мы настроили процесс web, который указывает Heroku, который Heroku использует для запуска веб-приложений, и в нем мы указали, как его запускать.
Также файл .gitignore я добавил, в котором, понятно, я указал, какие файлы мы игнорируем. Хотя давайте посмотрим нашу единственную view, это class-based view, унаследованный от template view. Он умеет показывать шаблон, и в контекст мы добавляем data['counter'], который возвращает наш класс counter. Он умеет соединяться с redis, используя переменные из settigns, и умеет инкрементить в redis счетчик и возвращать его. Также здесь есть шаблон, который выводит надпись "Hello, world" и текущее значение счетчика.

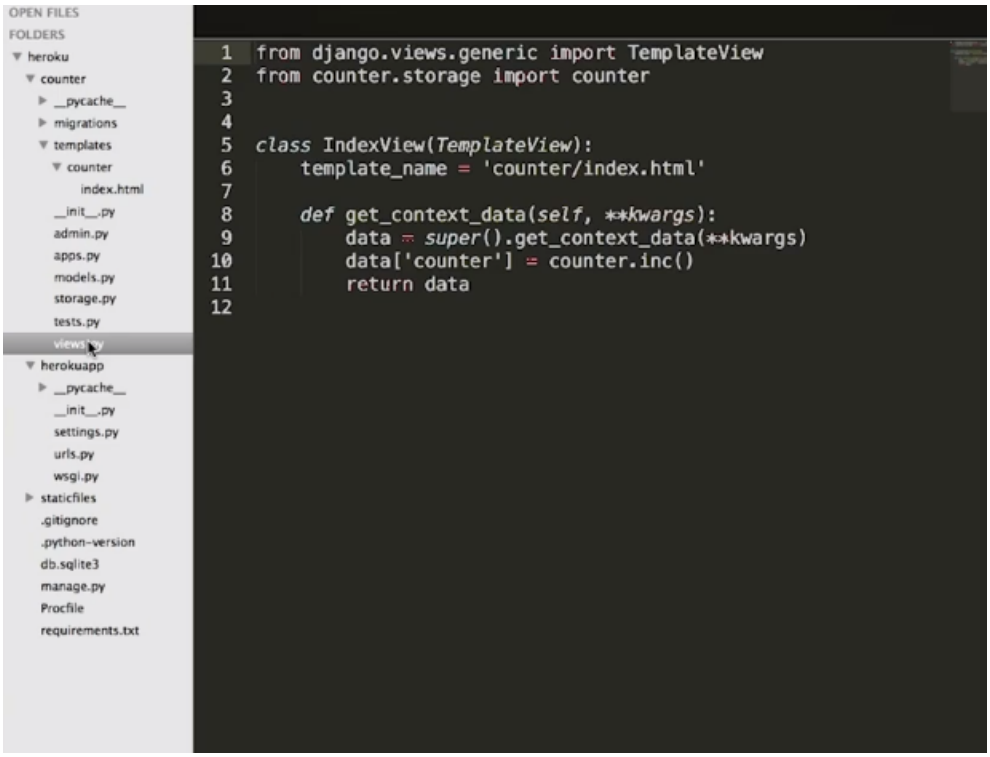

Рис. 21

В urls добавили url для нашего view, он показывается в корне, а в settings импортируем django-heroku. И внизу вызываем.

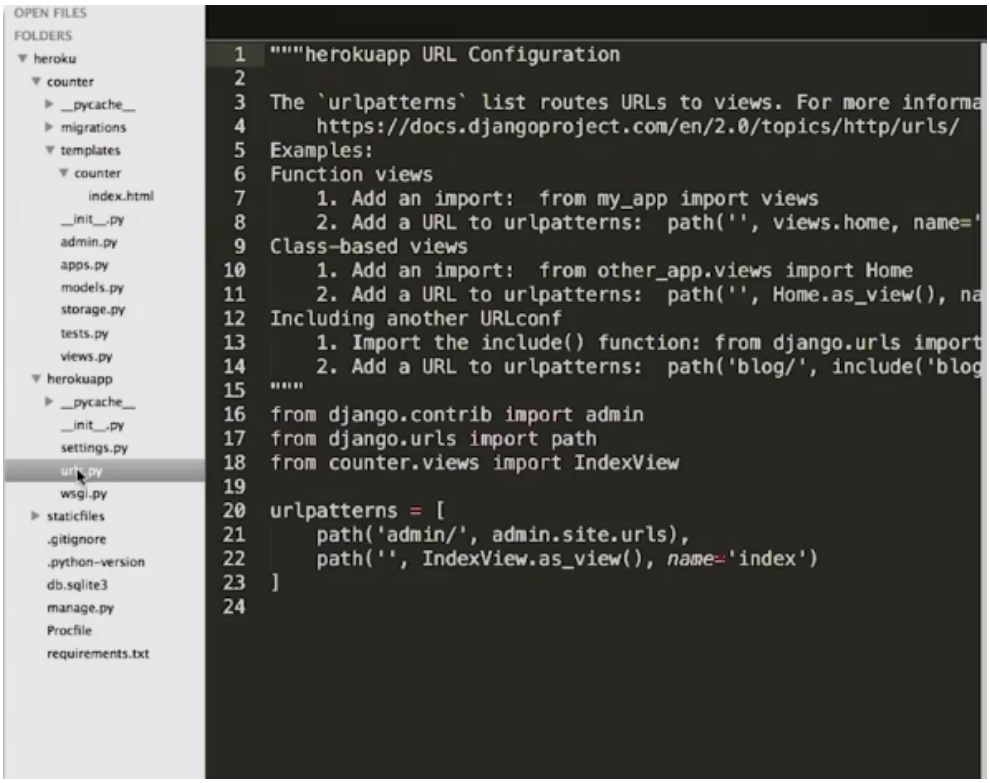

Рис. 22

Также смотрите: Heroku устанавливает ряд переменных окружения, плюс мы можем установить какие-то свои через интерфейс. А внутри settings мы можем их получить через os.getenv. В самом низу файла я вызываю heroku.settings, который помогает переконфигурировать базу данных, разрешенные хосты, раздачу статики, секретный ключ, взяв данные как раз из переменных окружения, выставленных Heroku.

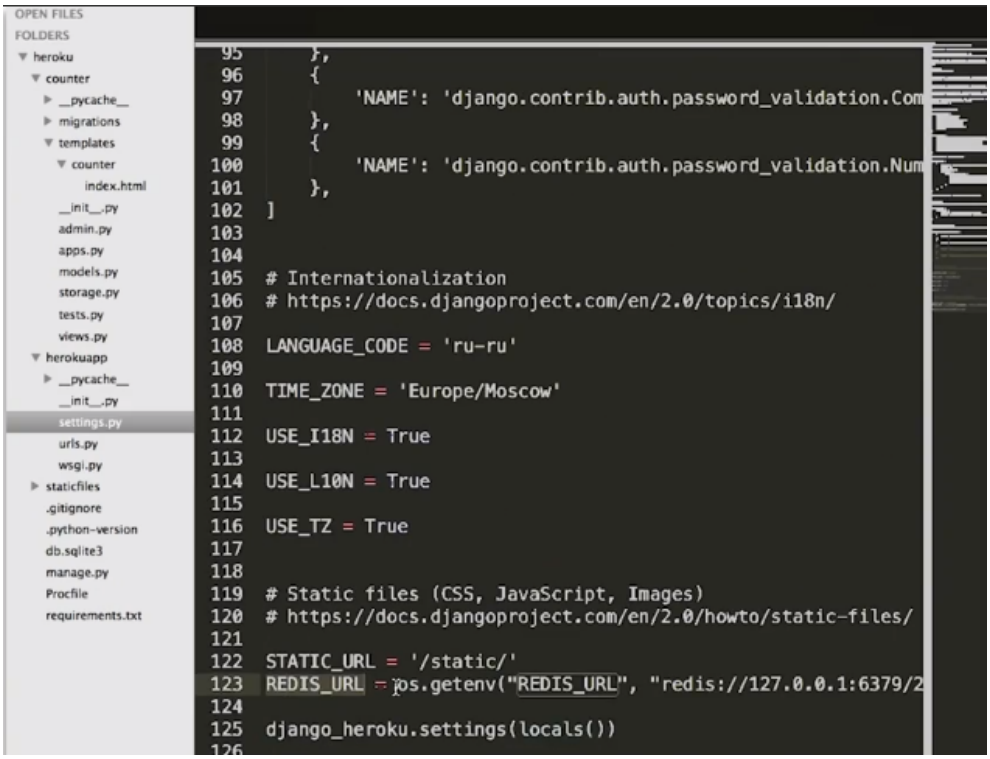

Рис. 23

Давайте разберемся с самим Heroku.
Во-первых, нам потребуется консольная утилита от Heroku. Их можно скачать с его сайта или поставить пакетным менеджером для вашей операционной системы.
Также нужно зарегистрироваться на Heroku и создать проект. Создать проект можно, здесь нажав на New, Create new app. Нужно в папке с вашим проектом добавить удаленный репозиторий heroku. Это можно сделать с помощью консольной утилиты Heroku. Чтобы разложить приложение, нам будет достаточно сделать просто push.
Но для работы приложения нам надо еще кое-что здесь сконфигурировать.
Во-первых, нашему приложению нужны некоторые ресурсы. В частности, это redis. Давайте начнем здесь писать слово redis, и вот есть Heroku Redis. Здесь можно выбрать тарифный план, но мы выберем бесплатный.

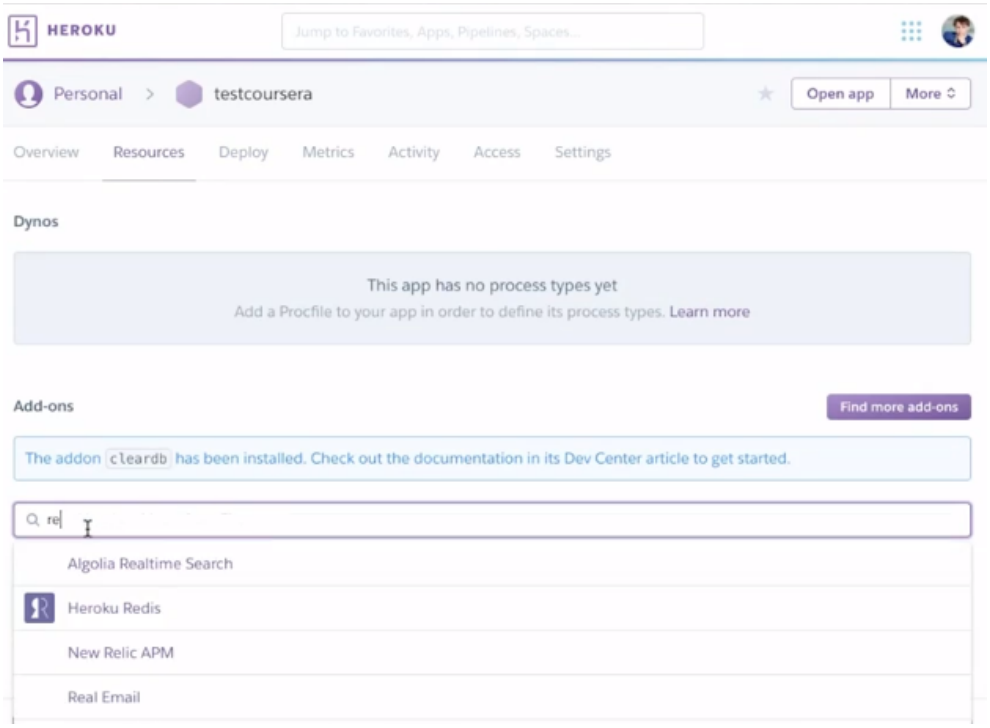

Рис. 24

Еще мы можем добавить какую-нибудь базу данных. Например, Postgres.
Сейчас идет provisioning, настройка Heroku Redis, а пока давайте переключимся в settings. Все установленные нами приложения добавили переменные окружения. Redis поставил свой url, DATABASE_URL поставил Postgres. Мы можем добавить свои, как я говорил уже, переменные, например, DEBUG. Мы оставим пустое, чтобы выключить DEBUG. И SECRET_KEY — это ключ, который Django использует для шифрования всяких своих данных, сессий и так далее.

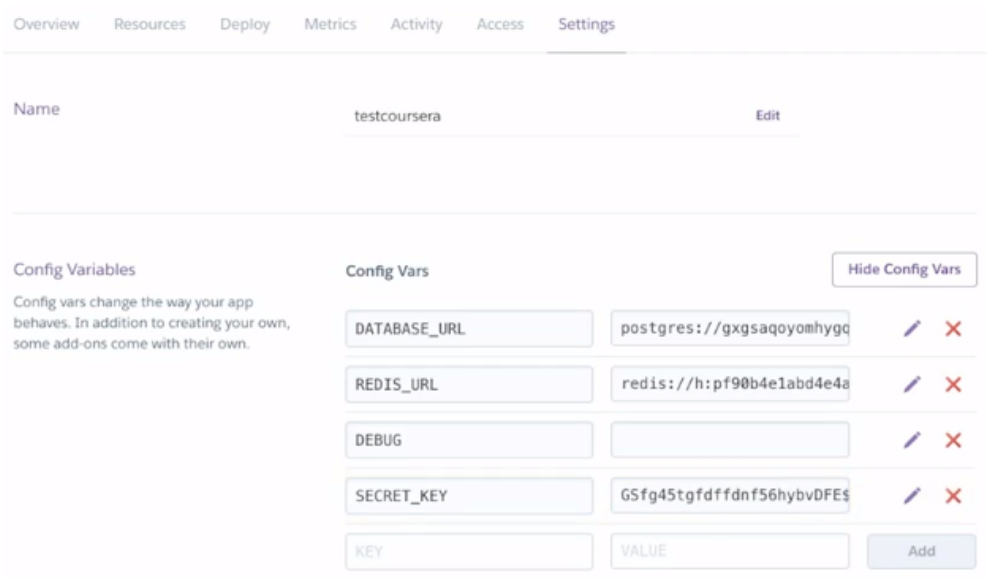

Рис. 25

Таким образом, мы добавили удаленный репозиторий еще один. Смотрите, если мы напишем git remote, то мы видим, что у нас есть origin, который у нас был, и heroku. И теперь, чтобы задеплоиться на heroku, нам достаточно сделать git push heroku master.

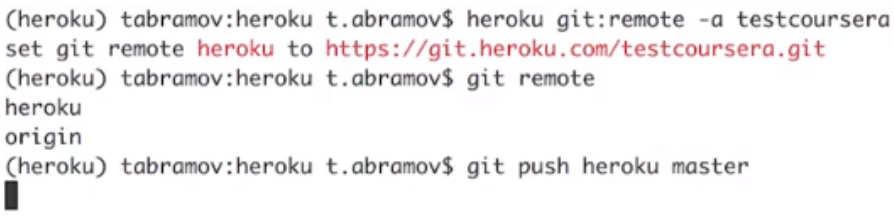

Рис. 26

В конце Heroku скажет, что приложение доступно и даже выдаст его url, по которому оно доступно всему миру.