



Создаем страницы для сайта на Django и публикуем на Heroku — Урок №3



В данном уроке мы создадим, протестируем и опробуем приложение Pages, у которого будет своя домашняя страница, а также страница с описанием проекта. Мы изучим классовые представления и **шаблоны Django**, что являются своеобразными кирпичиками в составе более сложных приложений.

Содержание статьи

- [Начальная настройка приложения в Django](#)

- [Шаблоны в Django](#)
- [Классовые представления в Django](#)
- [Настройка URL для приложений в Django](#)
- [Создание новой страницы «О нас» в Django](#)
- [Расширяем возможности шаблона в Django](#)
- [Пишем тесты для Django приложений](#)
- [Работа с Git и GitHub](#)
- [Локальный веб-сервер или Продакшн?](#)
- [Запускаем сайт на Django через Heroku](#)
- [Дополнительные файлы приложения Django](#)
- [Размещение Django сайта на Heroku](#)

Начальная настройка приложения в Django

Начальная **настройка приложения Django** включает следующие этапы:

- создание директории для кода;
- установка Django в новом виртуальном окружении;
- **создание нового проекта** в Django;
- создание нового приложения pages;
- обновление файла `settings.py`.

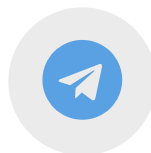
Будучи в **командной строке**, убедитесь, что вы сейчас не находитесь в действующем виртуальном окружении. Если перед знаком доллара (\$) есть текст в скобках, значит, окружение активно. Выполните команду `exit` и деактивируйте его.



Есть вопросы по Python?

На нашем форуме вы можете задать любой вопрос и получить ответ от всего нашего сообщества!

 Python Форум Помощи



Telegram Чат & Канал

Вступите в наш дружный **чат по Python** и начните общение с единомышленниками! Станьте частью большого сообщества!

 Чат

Канал



Паблик VK

Одно из самых больших сообществ по Python в социальной сети VK. **Видео уроки и книги** для вас!

 Подписаться

Создаем новую директорию под названием `pages` на рабочем столе или в любом другом месте. Главное, чтобы папка была легко доступной и не пересекалась с другими проектами.

В заново открытой командной строке наберите следующее:

```
1 $ cd ~/Desktop
2 $ mkdir pages && cd pages
3 $ pipenv install django==3.0.*
4 $ pipenv shell
5 (pages) $ django-admin startproject pages_project .
6 (pages) $ python manage.py startapp pages
```

Откройте в вашем текстовом редакторе файл `settings.py`. В самом низу списка проектов `INSTALLED_APPS` добавьте приложение `pages`:

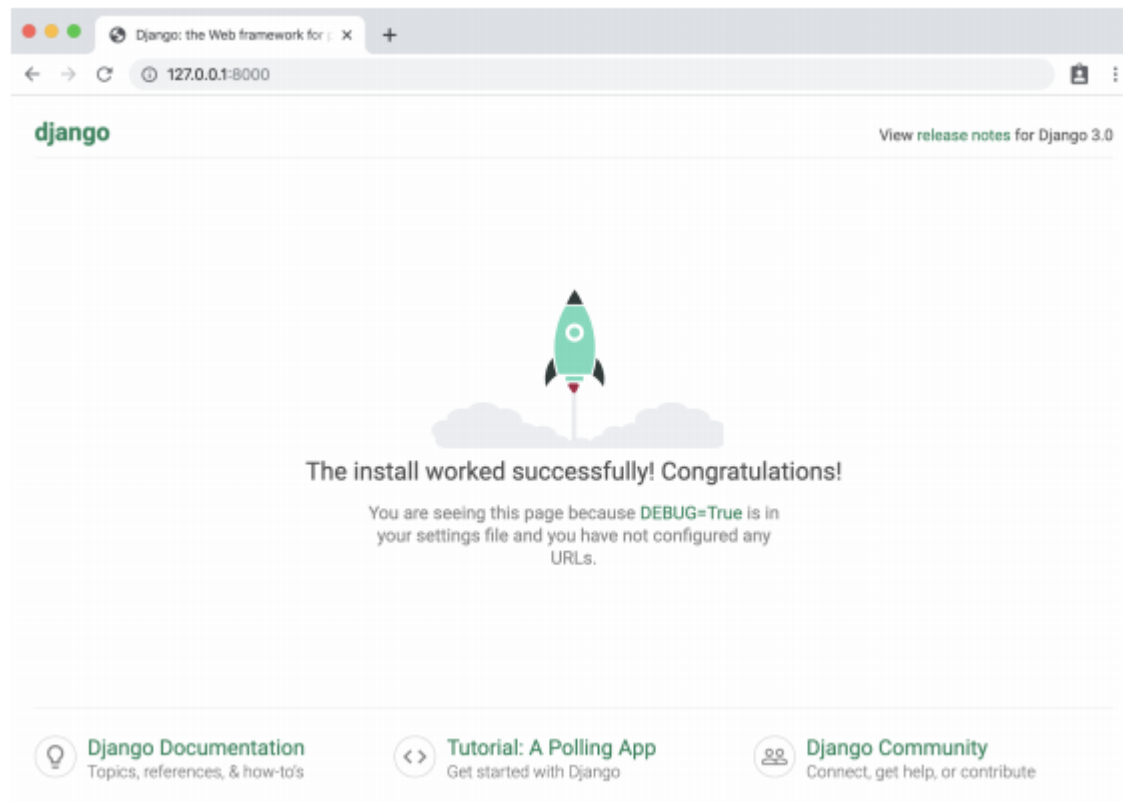
```
1 # pages_project/settings.py
2 INSTALLED_APPS = [
3     'django.contrib.admin',
4     'django.contrib.auth',
5     'django.contrib.contenttypes',
6     'django.contrib.sessions',
7     'django.contrib.messages',
8     'django.contrib.staticfiles',
9     'pages.apps.PagesConfig', # НОВОЕ
10 ]
```

Запускаем локальный сервер при помощи команды runserver:



A terminal window with a title bar containing icons for menu, back, forward, and search, followed by the text "Shell". The terminal shows the command: `(pages) $ python manage.py runserver`

Затем переходим на `http://127.0.0.1:8000/`.



Приветственная страница Django

Шаблоны в Django

Каждый веб-фреймворк нуждается в удобном способе генерации файлов HTML. В Django за это отвечают *шаблоны*: индивидуальные файлы HTML, которые связаны между собой и включают базовые логические операции.

Вспомним, что в предыдущей уроке на сайте «Hello, World» фраза была вписана сразу в код файла `views.py` как строка игнорируя какие либо **HTML шаблоны**. Технически это работает, но масштабируется не очень хорошо. Предпочтительнее будет связать представление (View) с шаблоном (Template), таким образом отделяя информацию из каждого.

В данном уроке мы научимся использовать шаблоны для **создания домашней страницы** и страницы с описанием проекта. В дальнейшем можно будет использовать шаблоны при создании сайтов с сотнями, тысячами и даже миллионами страниц,

используя при этом минимальное количество кода.

Для начала нужно определить, где поместить шаблоны внутри структуры проекта Django. Есть два варианта. По умолчанию загрузчик шаблонов Django осмотрит каждое приложение, выискивая связанные шаблоны. Тем не менее, структура остается несколько запутанной: каждое приложение нуждается в новой директории `templates`, другой директории с таким же названием, как и у приложения, а также в файле шаблона.

Следовательно, рассматривая приложение `pages`, Django будет ожидать следующую структуру:



```
1 └─ pages
2   └─ templates
3     └─ pages
4       └─ home.html
```

Отсюда следует, что нам нужно будет создать новую директорию `templates`, новую директорию с названием приложения `pages` и сам шаблон под названием `home.html`.



Зачем использовать этот, казалось бы, репетативный подход? Короткий ответ заключается в том, что **загрузчик шаблонов Django** должен наверняка найти подходящий шаблон! Что случится, если имеются два различных файла `home.html` в пределах двух различных приложений? Использование данной структуры гарантирует, что конфликтов подобного рода не произойдет.

Существует еще один подход к решению вопроса — это создание одной директории `templates` на уровне проекта и размещение там всех шаблонов. Сделав небольшие поправки в файле `settings.py`, можно указать Django, чтобы в поисках верного шаблона он рассматривал также эту новую директорию. Именно этот подход мы сейчас используем.

Первым делом покинем запущенный веб-сервер, применив комбинацию CTRL+C. Затем создадим директорию под названием `templates` и файл HTML под названием `home.html`.

```
1 (pages) $ mkdir templates
2 (pages) $ touch templates/home.html
```

После этого нам нужно обновить файл `settings.py` и указать Django место новой директории `templates`. Это изменение находится в одну строчку в настройках `'DIRS'` под `TEMPLATES`.

```
1 # pages_project/settings.py
2 TEMPLATES = [
3     {
4         ...
5         'DIRS': [os.path.join(BASE_DIR, 'templates')], # new
6         ...
7     },
8 ]
```

Затем для файла `home.html` добавляем обычный H1 заголовок.

```
1 <!-- templates/home.html -->
2 <h1>Homepage</h1>
```

Вот и все, шаблон готов! Следующим шагом будет конфигурация нашего URL и файлов представления (`views.py`).

Классовые представления в Django

Ранние версии Django поддерживали только **функциональные представления** (они же представления на основе функции), однако разработчики довольно быстро поняли, что им приходится повторять из раза в раз один и тот же паттерн. Писать представление,

которое составляет список всех объектов в модели. Писать представление, которое показывает только один детализированный элемент модели. И так далее.

Функциональные представления-генерики были введены для того чтобы избавить разработчиков от этих монотонных паттернов. Однако расширить возможности имеющихся представлений или изменить их должным образом оказалось не так уж просто. В результате в Django появились классовые представления-генерики, которые были легкими в использовании и давали возможность приспособливать имеющиеся представления под нужды наиболее часто встречающихся случаев.

Классы являются фундаментальной частью Python. В нашем представлении для отображения шаблона мы используем встроенный TemplateView. Обновим файл `pages/views.py`.

```
1 # pages/views.py
2 from django.views.generic import TemplateView
3
4
5 class HomePageView(TemplateView):
6     template_name = 'home.html'
```

Стоит отметить, что был применен стиль CamelCase для представления **HomePageView**, так как теперь он стал классом. Названия классов, в отличие от названий функций, всегда должны быть написаны с большой буквы. `TemplateView` уже содержит все логические операции, необходимые для отображения нашего шаблона, осталось только уточнить название шаблона.

Настройка URL для приложений в Django

На последнем этапе необходимо обновить **URLConfs**. Обновления требуется делать в двух местах. Первым делом обновляем файл самого проекта `pages_project/urls.py`, чтобы отметить наше приложение `pages`, а затем внутри приложения `pages` мы связываем представления (View) с URL-адресами.

Начнем с файла `pages_project/urls.py`.

```
1 # pages_project/urls.py
```

```
2 from django.contrib import admin
3 from django.urls import path, include # новое
4
5 urlpatterns = [
6     path('admin/', admin.site.urls),
7     path('', include('pages.urls')), # новое
8 ]
```

Данный код вам уже знаком. На второй строке мы добавляем `include`, что нужно для указания существующего URL-адреса для приложения `pages`. Затем создаем на уровне приложений файл `urls.py`.



```
1 (pages) $ touch pages/urls.py
```

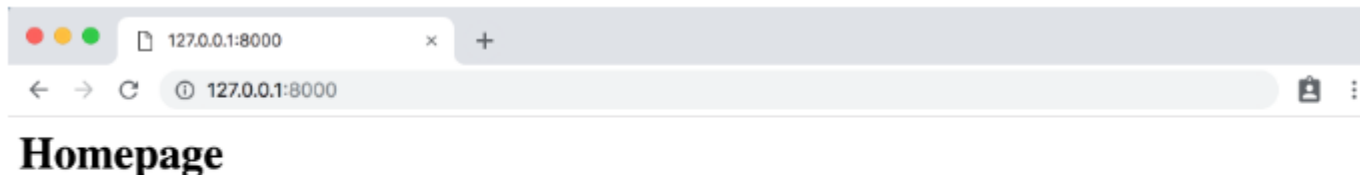
И затем добавляем следующий код.



```
1 # pages/urls.py
2 from django.urls import path
3
4 from .views import HomePageView
5
6 urlpatterns = [
7     path('', HomePageView.as_view(), name='home'),
8 ]
```

Данный паттерн практически полностью идентичен тому, что мы использовали в [прошлом уроке](#), однако есть одно важное отличие: во время использования классовых представлений в конце названия представления добавляется `as_view()`.

Вот и все! Теперь запустите веб-сервер через `python manage.py runserver`, а затем откройте `http://127.0.0.1:8000/`. Должна открыться новая домашняя страница.



Домашняя страница

Создание новой страницы «О нас» в Django

Процесс добавления страницы «О нас» очень похож на то, что мы только что сделали. Создадим новый файл шаблона, новое представление, а также новый адрес URL.

Закрываем веб-сервер через комбинацию CTRL+C и **создаем новый шаблон** под названием `about.html`.

```
(pages) $ touch templates/about.html
```

Затем добавляем короткий HTML заголовок.

```
<!-- templates/about.html -->
<h1>About page</h1>
```

Создаем новое представление для страницы.

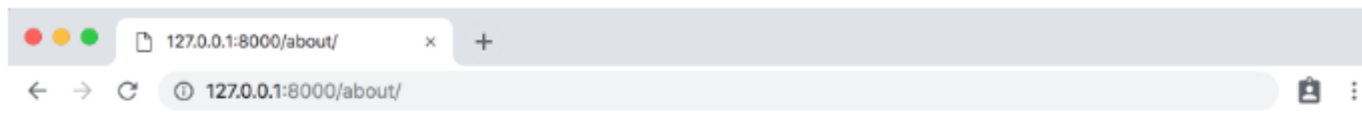
```
1 # pages/views.py
2 from django.views.generic import TemplateView
3
4 class HomePageView(TemplateView):
5     template_name = 'home.html'
6
7 class AboutPageView(TemplateView): # новое
8     template_name = 'about.html'
```

Далее связываем его с URL в about/.

```
1 # pages/urls.py
2 from django.urls import path
3
4 from .views import HomePageView, AboutPageView # новое
5
6 urlpatterns = [
7     path('about/', AboutPageView.as_view(), name='about'), # новое
8     path('', HomePageView.as_view(), name='home'),
9 ]
```

Запускаем сервер при помощи `python manage.py runserver`.

В браузере переходим по адресу `http://127.0.0.1:8000/about`. У вас должна появиться новая страница — «About page».



Страница «О Нас»

Расширяем возможности шаблона в Django

Главная сила шаблонов в их способности расширяться. Если задуматься, то на большинстве сайтов есть содержимое, которое повторяется на каждой странице (заголовки, футеры и так далее). Для разработчиков было бы здорово иметь одно установленное место для кода заголовка, которое бы передавалось по наследству каждому шаблону.

Это возможно! Создадим HTML-файл `base.html`, у которого будет заголовок с ссылками на две созданные нами страницы. Название для файла можно выбрать любое, в данной случае `base.html` просто стало традицией. Теперь закрываем веб-сервер `CTRL+C` и затем **создаем новый файл**.

```
1 (pages) $ touch templates/base.html
```

В Django, шаблонный язык для добавления ссылок и базовых логических операций минимален. Ознакомиться со встроенным списком шаблонных тегов можно в [официальной документации](#). **Шаблонные теги** оформляются так `{% something %}`, где «something» сам по себе является тегом. Вы даже можете **создать собственные шаблонные теги**.

Для добавления URL-ссылок в проект мы можем использовать [встроенный шаблонный тег `url`](#), который присваивает себе имя URL паттерна в качестве аргумента. Помните, как мы добавляли опциональные URL названия двум адресам в `pages/urls.py`? Это было сделано именно по этой причине. Тег `url` использует эти названия для автоматического создания ссылок.

URL путь для нашей домашней страницы был назван `home`, поэтому для настройки ссылки к ней мы будем использовать: `{% url 'home' %}`.

```
1 <!-- templates/base.html -->
2 <header>
3   <a href="{% url 'home' %}">Home</a> | <a href="{% url 'about' %}">About</a>
4 </header>
5
6 {% block content %}
7 {% endblock content %}
```

Внизу мы добавляем тег-блок под названием `content`. При наследовании, блоки могут быть переписаны дочерними шаблонами. Закрывающему тегу можно дать название `endblock` — просто напишите `{% endblock %}`. Это может стать хорошей подсказкой при ориентировке в крупных файлах шаблонов.

Обновим файлы `home.html` и `about.html` для расширения шаблона `base.html`. Это значит, что мы будем заново использовать код из одного шаблона в другой. Язык шаблонов в Django поставляется вместе с методом `extends`, который мы сейчас используем.

```
1 <!-- templates/home.html -->
2 {% extends 'base.html' %}
3
4 {% block content %}
5 <h1>Homepage</h1>
6 {% endblock content %}
```

и

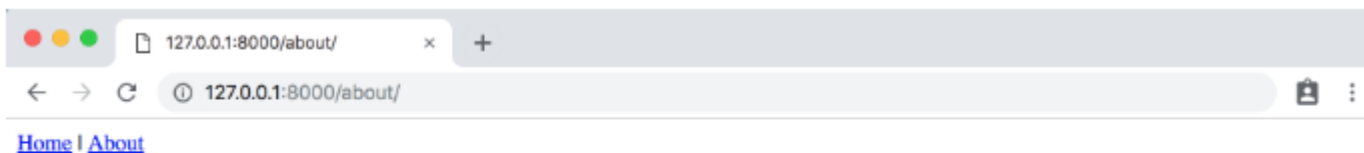
```
1 <!-- templates/about.html -->
2 {% extends 'base.html' %}
3
4 {% block content %}
5 <h1>About page</h1>
6 {% endblock content %}
```

Запускаем сервер с `python manage.py runserver` и открываем страницы `http://127.0.0.1:8000/` и `http://127.0.0.1:8000/about` вновь. Как видите, на обеих страницах появились заголовки.

Неплохо, правда?



Домашняя страница с заголовком



About page

Страница «About» с заголовком

С шаблонами можно осуществить множество операций. Обычно создается файл `base.html`, а затем в проекте добавляются дополнительные шаблоны которые расширяют базовый файл.

Пишем тесты для Django приложений

Наконец-то мы добрались до тестов. Хотя в приложениях это является базовым концептом, очень важно, чтобы добавление тестов в Django вошло у вас в привычку. Цитируя Джейкоба Каплан-Мосса, одного из создателей Django: «**Непротестированный код можно считать сломанным**».

Написание тестов важно, так как это автоматизирует процесс подтверждения того, что код работает должным образом. В приложении, подобном этому, вы сами можете увидеть, что домашняя страница и страница с описанием проекта на месте и содержат все необходимые данные. Однако с ростом нашего проекта, будет увеличиваться и количество веб-страниц, поэтому идея самостоятельной проверки сотен или даже тысяч страниц не представляется возможной.

Кроме того, когда мы делаем определенные изменения в коде — добавляем новый функционал, изменяем существующий, удаляем неиспользуемые элементы сайта — мы должны быть уверены в том, что все оставшиеся аспекты сайта по-прежнему работают должным образом. **Автоматические тесты** действуют таким образом, что компьютер каждый раз будет проверять работоспособность проекта, отталкиваясь от единожды написанного ранее правильного кода. К счастью, в Django есть встроенные инструменты для написания и запуска тестов.

```
1 # pages/tests.py
2 from django.test import SimpleTestCase
3
4
5 class SimpleTests(SimpleTestCase):
6     def test_home_page_status_code(self):
7         response = self.client.get('/')
8         self.assertEqual(response.status_code, 200)
9
```

```
10 def test_about_page_status_code(self):
11     response = self.client.get('/about/')
12     self.assertEqual(response.status_code, 200)
```

База данных нам пока не нужна, поэтому сейчас можно использовать простой SimpleTestCase. При наличии базы данных нужно обратиться к TestCase. Затем проводится проверка, в результате которой у каждой страницы должен быть код состояния 200 — это успешный ответ на стандартный HTTP запрос. Таким образом, становится понятно, что запрашиваемая страница действительно существует, но при этом не раскрывается ее содержимое.

Для **запуска теста** остановите веб-сервер, используя комбинацию CTRL+C, а затем наберите в командной строке `python manage.py test`:

```
(pages) $ python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
..
-----
Ran 2 tests in 0.014s
OK
Destroying test database for alias 'default'...
```

Все успешно! В будущем мы будем использовать более сложные тесты, особенно это важно при работе с **базами данных**.



А пока, важно привыкнуть к использованию тестов при каждом добавлении нового аспекта в существующий проект Django.


Работа с Git и GitHub

Пришло время зафиксировать изменения с git и загрузить данные на GitHub. Начнем с инициализации нашей директории.



```
1 (pages) $ git init
```

Используйте `git status` для просмотра изменений в коде, а затем `git add -A` для их добавления. Теперь мы можем добавить первый коммит.



```
1 (pages) $ git status
2 (pages) $ git add -A
3 (pages) $ git commit -m 'initial commit'
```

На GitHub создаем новое хранилище. Назовем его `pages-app`. Не забудьте отметить тип «Private», а затем нажмите кнопку «Create repository».

На следующей страницы пролистайте вниз до фразы «...or push an existing repository from the command line». Скопируйте текст и вставьте две команды в терминале.

Все должно выглядеть, так как указано ниже, только вместо `wsvincent` в качестве имени пользователя будет указано ваше никнейм на GitHub.



```
1 (pages) $ git remote add origin https://github.com/wsvincent/pages-app.git
2 (pages) $ git push -u origin master
```

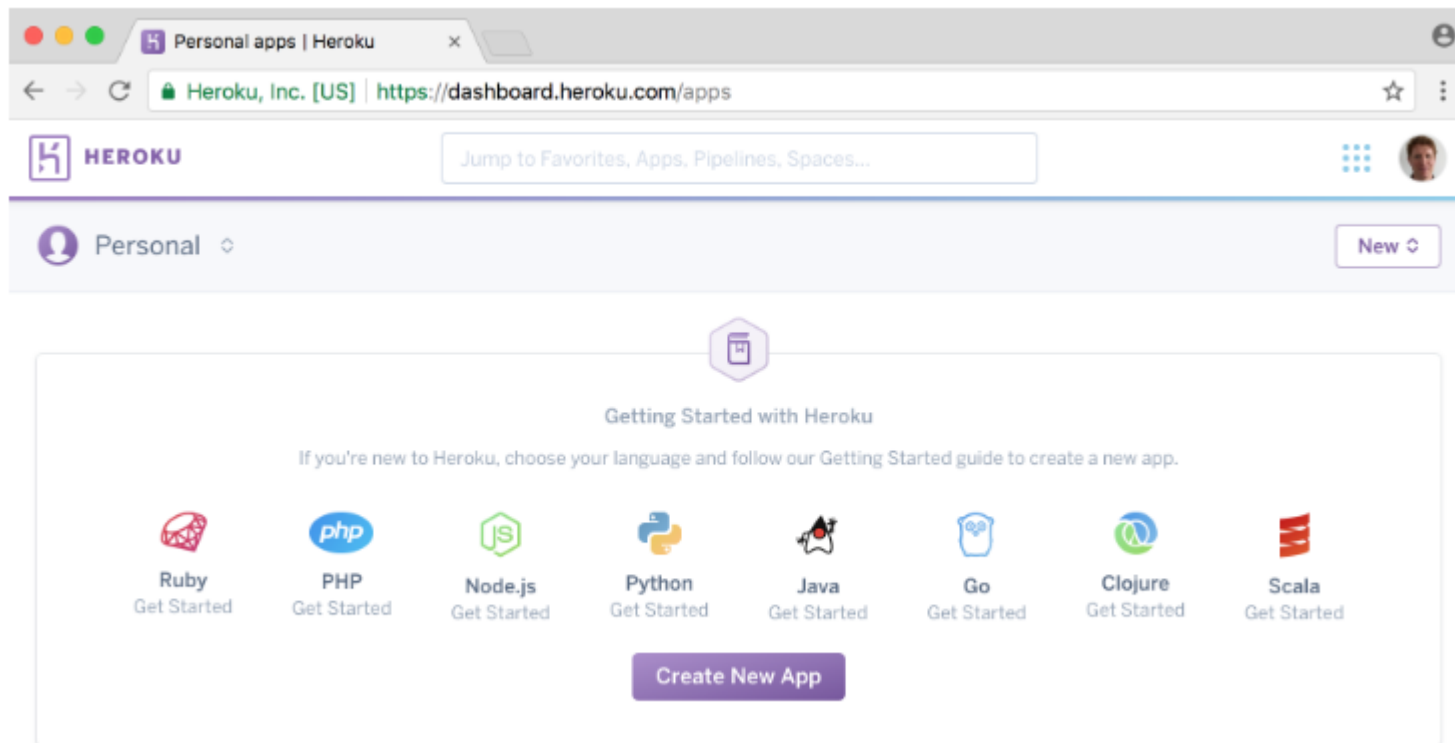
Локальный веб-сервер или Продакшн?

До этого момента, для запуска приложения *Pages* локально на компьютере мы использовали внутренний веб-сервер Django. Однако локальным адресом нельзя ни с кем поделиться. Для того чтобы запустить сайт в Интернете, сделав его доступным для всех, необходимо **разместить код на внешнем сервере**. Данный процесс получил английское название продакшн, под которым понимается процесс эксплуатации программы реальными людьми через интернет. Локальные данные доступны только на компьютере разработчика, а продакшн-код находится на внешнем сервере и становятся доступными для всех.

Существует великое множество серверов. Мы будем использовать Heroku. Для небольших проектов его можно использовать бесплатно, а несложный процесс размещения кода позволил Heroku завоевать большую популярность.

Запускаем сайт на Django через Heroku

Можете бесплатно зарегистрироваться на сайте Heroku. После подтверждения электронной почты вы будете перенаправлены на главную страницу.



Главная страница Heroku

Теперь необходимо установить Heroku *Command Line Interface* (CLI), необходимый для работы с командной строкой. Нам нужно установить Heroku глобально — таким образом он будет. Откройте новую вкладку командной строки при помощи комбинации CTRL+T, которая подходит как для Mac, так и для Windows.

Работая на Mac, в новой вкладке при помощи Homebrew установите Heroku:

```
1 $ brew install heroku/brew/heroku
```

Пользователям Windows нужно выбрать 32-битную или 64-битную версию установщика на странице загрузки [Heroku CLI](#). Для пользователей Linux на сайте Heroku предусмотрены [специальные инструкции для установки](#).

Установка Heroku на Ubuntu

```
1 sudo snap install --classic heroku
```

После завершения установки можете закрыть используемую вкладку и вернуться к главной вкладке с **активным виртуальным окружением** pages.

Наберите команду `heroku login`, после чего используйте только что установленные адрес электронной почты и пароль для Heroku.

```
1 (pages) $ heroku login
2 Enter your Heroku credentials:
3 Email: will@wsvincent.com
4 Password: *****
5 Logged in as will@wsvincent.com
```

Дополнительные файлы Django-приложения

Перед размещением кода на Heroku, нам понадобится сделать четыре изменения в нашем проекте *Pages*:

- обновить `Pipfile.lock`;
- создать новый файл `Procfile`;

- установить на нашем веб-сервере Gunicorn;
- изменить строчку в файле `settings.py`.

Внутри уже существующего файла `Pipfile` уточним используемую версию Python — в нашем случае 3.8. Для этого добавим в нижней части файла следующие две строчки.

Pipfile



```
1 [requires]
2 python_version = "3.8"
```

Далее запускаем `pipenv lock` для генерации подходящего `Pipfile.lock`.



```
1 (pages) $ pipenv lock
```

В действительности, в поисках информации о виртуальном окружении, Heroku заглядывает внутрь нашего `Pipfile.lock`. По этой причине здесь нам пришлось добавить настройку версии языка программирования.

Затем создаем `Procfile`, который является специфическим файлом конфигурации для Heroku.



```
1 (pages) $ touch Procfile
```

Откройте `Procfile` при помощи текстового редактора и добавьте следующее:



```
1 web: gunicorn pages_project.wsgi --log-file -
```

Это говорит о том, что нам надо использовать Gunicorn, что является сервером, подходящем для продакшена, в то время как собственный сервер Django работает только в локальном окружении. **Устанавливаем gunicorn** при помощи Pipenv.

```
1 (pages) $ pipenv install gunicorn==19.9.0
```

Конфигурация для веб-сервера находится в файле **wsgi.py**, который Django автоматически создает для каждого нового проекта. Он находится в основной папке нашего проекта. Поскольку наш проект называется `pages_project`, сам файл находится в `pages_project/wsgi.py`.

Последний шаг — небольшое изменение в файле `settings.py`. Прокрутите вниз до части `ALLOWED_HOSTS` и добавьте `'*'`. Результат должен получиться следующим:

```
1 # pages_project/settings.py
2 ALLOWED_HOSTS = ['*']
```

Параметр `ALLOWED_HOSTS` показывает, какие имена хостов/доменов наш **сайт на Django** может использовать. Это мера безопасности для предотвращения атак, которые возможны даже при, казалось бы, безопасных конфигурациях веб-сервера.

Однако мы использовали знак звездочки `*`, а это значит, что все домены являются приемлемыми. На сайте продакшн-уровня вместо этого вам пришлось бы составить ясный список допустимых доменов.

Для проверки изменений мы выполним команду `git status`, добавляем новые файлы и затем коммитим их:

```
1 (pages) $ git status
2 (pages) $ git add -A
```

```
3 (pages) $ git commit -m "New updates for Heroku deployment"
```

Теперь мы можем разместить код на GitHub, создав онлайн копию наших изменений в коде.

```
1 (pages) $ git push -u origin master
```

Размещение Django сайта на Heroku

Последний шаг — это фактическое **размещение кода на Heroku**. Если вы раньше настраивали сервер, вы будете поражены тем, как сильно Heroku упрощает данный процесс.

Весь процесс будет состоять из следующих этапов:

- создайте новое приложение на Heroku и вставьте в него наш код;
- настройте взаимодействие с git, то есть так называемый «hook» для Heroku;
- настройте приложение на игнорирование статических файлов;
- для активации приложения в онлайн режим, запустите сервер Heroku;
- посетите приложение, перейдя по предоставленному Heroku URL адресу.

В качестве первого шага можем **создать новое приложение Heroku**. Для этого в командной строке наберите `heroku create`. Heroku создаст случайное имя для нашего приложения, в моем случае это `fathomless-hamlet-26076`. Ваше название будет другим.

```
1 (pages) $ heroku create
2 Creating app... done, ● fathomless-hamlet-26076
3 https://fathomless-hamlet-26076.herokuapp.com/ |
4 https://git.heroku.com/fathomless-hamlet-26076.git
```

На данный момент нам остается только настроить Heroku. Для этого укажем Heroku проигнорировать статические файлы вроде CSS и JavaScript, которые Django по умолчанию попытается исправить под себя. Выполним следующую команду:



```
(pages) $ heroku config:set DISABLE_COLLECTSTATIC=1
```

Теперь можем разместить код на Heroku.



```
(pages) $ git push heroku master
```

Если бы мы только что набрали `git push origin master`, то код был бы загружен в GitHub, а не в Heroku. Добавление слова `heroku` в команду позволяет отправить код на Heroku. Первые несколько раз это может сбивать с толку.


Наконец, нам нужно запустить в онлайн наше Heroku приложение. Поскольку трафик веб-сайтов растет, они нуждаются в дополнительных услугах от Heroku. Однако для нашего основного примера мы можем использовать самый низкий уровень `web=1`, который является бесплатным.

Введите следующую команду.



```
(pages) $ heroku ps:scale web=1
```

Все готово! Последним шагом станет подтверждение того, что наше приложение действительно запущено и работает в режиме онлайн. Если вы выполните команду `heroku open`, ваш браузер откроет новую вкладку с URL адресом вашего приложения:



```
(pages) $ heroku open
```

Наш адрес <https://fathomless-hamlet-26076.herokuapp.com/>. Вы можете убедиться в этом, вот появившаяся домашняя страница:



Домашняя страница на Heroku

Страница «About» также открылась должным образом:

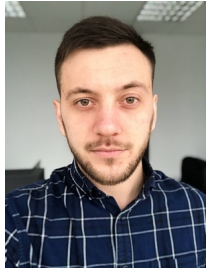


Страница «About» на Heroku

Вам нет нужды разлогиниваться или покидать свое приложение в Heroku. Оно будет работать само по себе на этом бесплатном уровне.

Заключение

Поздравляю с созданием и размещением своего первого Django проекта в интернете! Мы использовали шаблоны, классовые представления, изучили **URLConfs** подробнее, добавили базовые тесты и использовали Heroku для публикации нашего сайта.



Vasile Buldumac

Являюсь администратором нескольких порталов по обучению языков программирования Python, Golang и Kotlin. В составе небольшой команды единомышленником, мы занимаемся популяризацией языков программирования на русскоязычную аудиторию. Большая часть статей была адаптирована нами на русский язык и распространяется бесплатно.

E-mail: vasile.buldumac@ati.utm.md

Образование

Universitatea Tehnică a Moldovei (*utm.md*)

- 2014 — 2018 Технический Университет Молдовы, ИТ-Инженер. Тема дипломной работы «Автоматизация покупки и продажи криптовалюты используя технический анализ»
- 2018 — 2020 Технический Университет Молдовы, Магистр, Магистерская диссертация «Идентификация человека в киберпространстве по фотографии лица»

in

Ваше мнение о данной статье?

31 Ответов



Отлично!



Это интересно



Админ удали статью



Не интересно

0 Комментариев

python-scripts



Политика конфиденциальности Disqus

1

Войти ▾



Рекомендовать



Твитнуть



Поделиться

Лучшее в начале ▾



Начать обсуждение...

ВОЙТИ С ПОМОЩЬЮ

ИЛИ ЧЕРЕЗ DISQUS (?)

Имя

Прокомментируйте первым.



Подписаться



Добавь Disqus на свой сайт

Добавить DisqusДобавить



Do Not Sell My Data

Изучаем Python 3 на примерах

[Декораторы](#)[Уроки Tkinter](#)[Уроки PyCairo](#)[Установка Python 3 на Linux](#)[Контакты](#)

