Шаблон + Контекст = html

Паттерн MVC (или MVT - model view template). Разбираем понятие "свободное связывание". Работа со статикой. Шаблонизатор django (шаблонные фильтры, наследование шаблонов).

[Шаблоны и контекст](#_c713xnuw3xem)

[Представления](#_1ph11ss8rzdr)

[Патерн MVC](#_6es15eloc6lc)

[Работа со статикой](#_9kf802hsgu9r)

[Шаблонизатор Django](#_yc53f5hxs8hr)

[Шаблонные фильтры](#_huddx6twu6bm)

[Наследование шаблонов](#_v79to0ru2aot)

[Домашнее задание](#_owpsqd9opgcf)

[Дополнительные материалы](#_y937sk8fclye)

[Используемая литература](#_uvp6qax5r1ok)

# 

# Шаблоны и контекст

Будучи веб фреймверком, Django позволяет динамически генерировать HTML. Самый распространенный подход - использование шаблонов. Шаблоны содержат статический HTML и динамические данные, рендеринг которых описан специальным синтаксисом. Эти данные и называются **контекстом.**

Проект Django может использовать один или несколько механизмов создания шаблонов (или ни одного, если вы не используете шаблоны). Django предоставляет бэкенд для собственной системы шаблонов, которая называется язык шаблонов Django (Django template language, DTL),

Django предоставляет стандартный API для загрузки и рендеринга шаблонов, независимо от используемого бэкенда. Загрузка включает в себя поиск шаблона по названию и предварительную обработку, обычно выполняется загрузка шаблона в память. Рендеринг означает передачу данных контекста в шаблон и возвращение строки с результатом.

Язык шаблонов Django – собственная система шаблонов Django. До Django 1.8 – это была единственная альтернатива. Это хорошая система шаблонов, но со своими особенностями.

В качестве практического примера, дополним статических html страницы проекта “About me” данными ФИО автора. Т.к. у нас пока нет БД, то данные будем хранить в переменных в python-коде.

Допустим, на данный момент ваше ФИО на странице располагается так:

|  |
| --- |
| <div class="name">  <h1>  Юрченко Евгений Витальевич  </h1>  </div> |

Добавим данные в **views.py**

|  |
| --- |
| def main(request):  name = 'Евгений'  surname = 'Юрченко'  middle\_name = 'Витальевич'  return render\_to\_response("index.html") |

Чтобы передать эти данные в виде контекста в шаблон:

|  |
| --- |
| def main(request):  name = 'Евгений'  surname = 'Юрченко'  middle\_name = 'Витальевич'  return render\_to\_response("index.html", {'name': name, 'surname': surname, 'middle\_name': middle\_name}) |

Просто передаем их в виде словаря, вторым аргументом функции render\_to\_response

Осталось вставить данные в шаблон:

|  |
| --- |
| <div class="name">  <h1>  {{ surname }} {{ name }} {{ middle\_name }}  </h1>  </div> |

Используем шаблонные теги {{ <ключ словаря контекста> }}, вместо {{ name}} будет подставлено значение словаря контекста под ключем “name”.

Запустим сервер разработки, и проверим результат.

Django-шаблонизатор - это очень мощный инструмент, подробнее с ним будем знакомиться на протяжении всех уроков. Полную информацию можете посмотреть [здесь](http://djbook.ru/rel1.8/ref/templates/api.html).

# Представления

Теперь подробнее разберемся с функциями представления, располагающимися в файле views.py.

Каждое представление отвечает за выполнение одной из двух вещей: возвращение объекта HttpResponse, представляющего содержимое запрашиваемой страницы, или вызов исключения такого как Http404.

Как правило, представление извлекает данные в соответствии с заданными параметрами, загружает шаблон и отображает этот шаблон вместе с полученными данными.

Не забывайте, каждая функция представления первым аргументом получает объект запроса. Этот объект несет всю необходимую информацию о запросе.

Django использует объекты ответа и запроса, чтобы передавать состояние в системе.

Когда запрашивает страница, Django создает объект HttpRequest, который содержит различные данные о запросе. Потом Django определяет и загружает необходимое представление и вызывает его, передавая объект HttpRequest первым аргументом. Каждое представление должно вернуть объект HttpResponse.

Т.е. наша функция render\_to\_response() - всего лишь удобная обёртка для формирования ответа в соответствии с [протоколом http](https://ru.wikipedia.org/wiki/HTTP). Чтобы браузер корректно обработал ответ, ответ необходимо правильно сформировать, добавив тело, заголовки и прочую техническую информацию, эту рутинную операцию django берет на себя, предоставляя нам удобные инструменты.

Функции представления содержат бизнес-логику приложения (сайта), в то время как шаблоны содержат логику отображения. Если говорить проще: шаблон знает, как себя правильно отобразить, а функция представления знает, как получить правильный контекст (данные) для шаблона. В свою очередь модели (речь о которых пойдет в следующих уроках) описывают структуру хранения данных.

# Патерн MVC

Патерн MVC (или MVT - model view template).

Django поощряет свободное связывание и строгое разделение частей приложения. Если следовать этой философии, то легко вносить изменения в одну конкретную часть приложения без ущерба для остальных частей. В функциях представления, например, мы обсуждали важность отделения бизнес-логики от логики отображения с помощью шаблонной системы. Используя слой для работы с базой данных, мы применяем эту же философию для логики доступа к данным.

Эти три вещи вместе — логика доступа к данным, бизнес-логика и логика отображения — составляют концепцию, которую называют шаблоном Модель-Представление-Управление (Model-View-Controller, MVC) архитектуры программного обеспечения. В этой концепции термин «Модель» относится к логике доступа к данным; термин «Представление» относится к той части системы, которая определяет, что показать и как; а термин «Управление» относится к той части системы, которая определяет какое представление надо использовать, в зависимости от пользовательского ввода, по необходимости получая доступ к модели.

Почему используется сокращение?

Целью чёткого определения сокращений, подобных MVC, является упорядочивание взаимодействия между разработчиками. Вместо того, чтобы сказать вашим сотрудникам: «Давайте использовать абстрактный доступ к данным, затем создадим слой управления отображением данных и тогда создадим слой между ними, который всем этим управляет» можно воспользоваться общим термином: «Давайте здесь использовать подход MVC».

Django следует модели MVC достаточно близко, т.е., может быть назван MVC совместимой средой разработки. Вот примерно как M, V и C используются в Django:

* M, доступ к данным, обрабатывается слоем работы с базой данных, который описан в этой главе.
* V, эта часть, которая определяет какие данные получать и как их отображать, обрабатывается представлениями и шаблонами.
* C, эта часть, которая выбирает представление в зависимости от пользовательского ввода, обрабатывается самой средой разработки, следуя созданной вами схемой URL, и вызывает соответствующую функцию Python для указанного URL.

Так как «C» обрабатывается средой разработки и всё интересное в Django происходит в моделях, шаблонах и представлениях, на Django ссылаются как на MTV-ориентированную среду разработки. В MTV-подходе к разработке:

* M определено для «Модели» (Model), слоя доступа к данным. Этот слой знает всё о данных: как получить к ним доступ, как проверить их, как с ними работать и как данные связаны между собой.
* T определено для «Шаблона» (Template), слоя представления данных. Этот слой принимает решения относительно представления данных: как и что должно отображаться на странице или в другом типе документа.
* V определено для «Представления» (View), слоя бизнес-логики. Этот слой содержит логику, как получать доступ к моделям и применять соответствующий шаблон. Вы можете рассматривать его как мост между моделями и шаблонами.

Если вам приходилось работать с другими MVC ориентированными средами разработки, такими как Ruby on Rails, вы можете рассматривать представления в Django как «контролёры», а шаблоны Django — как «представления». Это печальная путаница возникла в результате различных толкований MVC. В интерпретации Django «представление» описывает данные, которые будут представлены пользователю. Неважно как эти данные будут выглядеть, важно какие данные. Напротив, в Ruby on Rails и подобных ему средах предполагается, что в работу контролёра включено принятие решения, какие данные будут представлены пользователю, в то время как представление точно определяет, как эти данные будут выглядеть, а не какие данные будут представлены.

Ни одна интерпретация не имеет преимуществ над другой. Важно понимать основную концепцию.

# Работа со статикой

Думаю, одной из проблем, при выполнении домашнего задания была именно работа со статикой.

Веб-приложения обычно требуют различные дополнительные файлы для своей работы (изображения, CSS, Javascript и др.). В Django их принято называть “статическими файлами” (прим. - или “статика”).

Вообще, в реальном “боевом” проекте django не занимается раздачей статики, это забота веб-сервера (nginx, apache и прочих), а пока в качестве такого веб-сервера выступает тестовый-джанго-сервер.

Пока не будем глубоко вникать с особенности работы со статикой и просто заставим сервер отдавать статические файлы при запросе браузера (именно браузер запрашивает эти файлы, встречая в html коде ссылки на них).

Создадим папку static в корне проекта для хранения всех статических файлов.

В конец settings.py Добавим код:

|  |
| --- |
| STATICFILES\_DIRS = (  os.path.join(BASE\_DIR, "static"),  ) |

**Обратите внимание:** внимательно изучите глобальную переменную BASE\_DIR, это хороший способ получения абсолютного пути проекта, который я бы советовал использовать в прочих проектах, чтобы избежать путаницы с путями внутри самого проекта.

Вот и всё, теперь статика отдается. Подробнее про работу со статическими файлами можете почитать [здесь](http://djbook.ru/rel1.6/howto/static-files/index.html).

# Шаблонизатор Django

Мы уже использовали шаблон переменной {{ }}. Пришло время подробнее познакомиться с шаблонизатором.

Теги выглядят таким образом: {% tag %}. Теги сложнее, чем переменные: одни создают текст для вывода, влияют на выполнение используя условия и циклы, другие загружают дополнительную информацию в шаблоны, чтобы использовать её далее через переменные.

Некоторые теги требуют открывающий и закрывающий теги (например, {% tag %} ... содержимое тега ... {% endtag %}).

Django содержит около двадцати встроенных тегов. Подробно обо всех тегах [здесь](http://djbook.ru/rel1.8/ref/templates/builtins.html#ref-templates-builtins-tags). Рассмотрим несколько для знакомства на примере нашего проекта.

На данный момент ваша страница со списком мест, где вы работали, выглядит подобным образом:

|  |
| --- |
| <h1>Опыт работы</h1>  <ol>  <li>Skyparkcdn</li>  <li>tranio.ru</li>  <li>НУДО "НАДиП</li>  <li>ООО «Терминал-Сервис»</li>  </ol> |

Понятно, для того, чтобы добавить или изменить одно из мест работы, вам придется менять код html страницы, а это совсем не то, чего мы хотим. Немного подумав, мы можем сделать так:

views.py

|  |
| --- |
| def works(request):  place\_of\_work = ['Skyparkcdn', 'tranio.ru', 'НУДО "НАДиП', 'ООО «Терминал-Сервис»']  return render\_to\_response("works.html", {'place\_of\_work': place\_of\_work}) |

Используя шаблон переменных, можно сделать так:

works.html

|  |
| --- |
| <h1>Опыт работы</h1>  <ol>  <li>{{ place\_of\_work.0 }}</li>  <li>{{ place\_of\_work.1 }}</li>  <li>{{ place\_of\_work.2 }}</li>  <li>{{ place\_of\_work.3 }}</li>  </ol> |

Уже лучше, но всё равно не то, что нам требуется (добавление нового места работы не добавит запись в списке работ).

Здесь отлично подойдут шаблонные теги цикла:

works.html

|  |
| --- |
| <h1>Опыт работы</h1>  <ol>  {% for work in place\_of\_work %}  <li>{{ work }}</li>  {% endfor %}  </ol> |

Очень похоже на python цикл for in.

Всё, наша задача решена. Наш шаблон отобразит любое количество мест работ из списка, нам больше не нужно изменять код шаблона, если изменятся данные.

Если список с метами работ будет пуст, мы просто ничего не увидим, а, например, мы бы хотели увидеть надпись “пока нигде не работал”, если мест работы нет.

Это легко сделать, используя шаблонный тег {% if %}

works.html

|  |
| --- |
| <h1>Опыт работы</h1>  <ol>  {% if place\_of\_work %}  {% for work in place\_of\_work %}  <li>{{ work }}</li>  {% endfor %}  {% else %}  Пока нигде не работал  {% endif %}  </ol> |

Надеюсь, комментарии излишни.

# Шаблонные фильтры

Допустим, вы хотите быть уверены, что ваши ФИО всегда будут выводиться с большой буквы, даже если вы где-то опечатаетесь (например, по невнимательности, напишите имя с маленькой буквы). Вы, конечно, можете добавить код по views, который будет устранять эту досадную проблему. Но это будет неверный подход, т.к. во views у вас появляется код, который влияет только на то, как будет отображена информация. Ведь мы знаем, что за отображение будет отвечать шаблон. Эту проблему легко можно решить, используя шаблонные фильтры.

Допускаем опечатку во views.py

|  |
| --- |
| def main(request):  name = 'евгений'  surname = 'Юрченко'  middle\_name = 'витальевич'  return render\_to\_response("index.html", {'name': name, 'surname': surname, 'middle\_name': middle\_name}) |

Добавляем фильтр в index.html

|  |
| --- |
| <div class="name">  <h1>  {{ surname|title }} {{ name|title }} {{ middle\_name|title }}  </h1>  </div> |

Шаблонные фильтры указываются через вертикальную черту (|). В примере используется фильтр title, который преобразует первый символ слов в верхний регистр, остальные в нижний.

Полный перечень [встроенных шаблонных фильтров](http://djbook.ru/rel1.8/ref/templates/builtins.html#ref-templates-builtins-filters). Если вы не найдете подходящего вам фильтра, довольно просто можно создать свой собственный. По сути, шаблонный фильтр, это просто функция Python (подробнее [здесь](http://djbook.ru/rel1.4/howto/custom-template-tags.html)).

# Наследование шаблонов

На данный момент наш проект состоит из трёх страниц, но все эти страницы являются частями одного сайта, и, скорее всего, будут иметь общие элементы, например, ко всем трём страницам подключена одна и та же таблица стилей, и все три страницы используют один и тот же header и footer. Хотя эти элементы полностью идентичны, если вы решите изменить footer, вам придётся менять его сразу в трёх местах (это только пока страницы ТРИ).

Чтобы избежать дублирования кода в шаблонах и выделить общие элементы, используется механизм наследования шаблонов.

В сущности, наследование шаблонов позволяет создать основной «скелетный» шаблон, который содержит все общие части вашего сайта и определить «блоки», которые могут быть заменены шаблонами-наследниками.

Определяем базовый шаблон

base.html

|  |
| --- |
| {#Любая страница проекта должна наследоваться от данной страницы, или от ее потомков#}  <!DOCTYPE html>  <html>  <head>  <title>  {% block title %}  BASE TEMPLATE  {% endblock %}  </title>  {% block css %}  {#Тут подключаем общие для всего сайта таблицы стилей #}  <link rel="stylesheet" href="/static/css/style.css"/>  {% endblock %}  <meta charset="utf-8">  </head>  <body>  <div class="container">  <div class="page-header">  <div class="row">  {% block header %}  <h1>"About me"</h1>  {% endblock %}  </div>  </div>  <div class="row">  {% block menu %}  |<a href="/works">Работа</a>|<a href="/teach">Учеба</a> |  {% endblock %}  </div>  <div class="row">  {% block content %}  Текст базового шаблона(просто для демонстрации)  {% endblock %}  </div>  <div class="modal-footer">  {% block footer %}  &copy; 2016 Booblegum  {% endblock %}  </div>  </div>  </body>  </html> |

В данном шаблоне описываем элементы общие для всех страниц сайта.

Теперь можно наследоваться от данной страницы и переопределить содержимое нужных блоков.

index.html

|  |
| --- |
| {% extends "base.html" %}  {% block content %}  <h1>Welcome</h1>  <div class="name">  <h1>  {{ surname|title }} {{ name|title }} {{ middle\_name|title }}  </h1>  </div>  {% endblock %} |

Содержимое блока content будет заменено, а содержимое остальных блоков останется без изменений.

Здесь мы используем тег{% block %}. Всё, что такие теги делают — указывают шаблонной системе, что дочерние шаблоны могут переопределять эту часть основного шаблона.

Ниже приведены некоторые советы для работы с наследованием шаблонов:

* Если вы используете {% extends %} в шаблоне, он должен быть первым тегом в этом шаблоне. В противном случае, наследование шаблонов работать не будет.
* В общем случае, чем больше тегов {% block %} в основном шаблоне, тем лучше. Запомните, дочерние шаблоны не обязаны определять все блоки основного шаблона. Так что вы можете указать разумные значения по умолчанию в ряде блоков, а затем определить в дочернем шаблоне только те, которые надо изменить. Лучше иметь больше обработчиков.
* Если вы заметили, что повторяете код в ряде шаблонов, возможно, вам надо перенести этот код в {% block %} в основном шаблоне.
* Если вам надо получить содержимое блока из основного шаблона, переменная {{ block.super }} поможет с этим. Это полезно, если вам потребуется лишь добавить данные в блок вместо его полной замены.
* Вы не можете определять множество тегов {% block %} с одинаковым именем в одном шаблоне. Это ограничение существует по причине того, что эти теги работают в «обоих» направлениях. То есть, этот тег не просто предоставляет место для заполнения данными, он также определяет содержимое, которое заполняет место в основном шаблоне. Если бы было два одинаково названных тега {% block %} в шаблоне, то основной шаблон бы не знал, какое содержимое блока использовать.
* В большинстве случаев, аргументом для тега {% extends %} будет строка. Но может быть и переменная, если вы не знаете имя основного шаблона до запуска приложения. Это позволяет вам реализовывать динамические вещи.

# 

# 

# Домашнее задание

Продолжаем дорабатывать проект About ME, используя новые знания, полученные на текущем вебинаре, а также пользуясь документацией по django (все ссылки в “Дополнительных материалах”)

1. Используя наследование шаблонов, выделить общие элементы вашего сайта в родительский шаблон base.html, наследуйте все текущие страницы от него.
2. Все данные о себе (ФИО, хобби, места работы и учебы) вынести из шаблонов, сохранив в переменных Python в удобной вам структуре. Проверить, чтобы все отображается корректно, если в python-коде добавить новые данные (например, добавить новое хобби или удалить несколько старых). Также добавить корректное отображение в случае отсутствия данных.
3. Продолжить работать над визуальной частью сайта (например, добавить иконки для хобби и/или добавить ссылки на сайты учебных заведений, в которых вы учились).
4. \*(усложненное) Используя дополнительные материалы, установите глобальную переменную STATIC\_URL, чтобы ваши пути выглядели как href="{{ STATIC\_URL }}css/base.css". Это позволит вам быстро изменить место храниения статических файлов, в случае необходимости.
5. \* Дату вашего рождения в контексте сохраните в виде date(). А в шаблоне выведите в удобном виде, используя шаблонный фильтр date.

Все проблемы и пути их решения, подробно обсудим на следующем занятии.

# Дополнительные материалы

Все то, о чём сказано в методичке, но подробнее:

1. [API Шаблонизатора](http://djbook.ru/rel1.8/ref/templates/api.html)
2. [Подробнее про статику](http://djbook.ru/rel1.6/howto/static-files/index.html)
3. [Наследование шаблонов](http://djbook.ru/ch04s07.html)
4. [Встроенные шаблонные фильтры](http://djbook.ru/rel1.8/ref/templates/builtins.html#ref-templates-builtins-filters)
5. [Встроенные шаблонные теги](http://djbook.ru/rel1.8/ref/templates/builtins.html#ref-templates-builtins-tags)

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [Официальная документация](https://docs.djangoproject.com/en/1.9/)
2. [Перевод документации](http://djbook.ru/)