

Веб-розробка з Python та Django для Початківців

*Виключно практичний посібник
по веб-розробці з допомогою мови Python
та фреймворку Django.
Для початківців.*



Віталій Подоба

Веб-розробка з Python та Django для Початківців

Виключно практичний посібник по
веб-розробці з допомогою мови Python та
фреймворку Django. Для початківців.

Віталій Подоба

©2014 - 2015 Віталій Подоба

Зміст

1. Вступ	1
Для кого і для чого дана книга?	1
Як працювати із даною книгою?	4
Які технології розглянемо у даній книзі?	8
Організовуємось!	9
Мотивуємось!	13
Домашнє завдання	16
2. Що таке веб-розробка?	18
Комунікація Клієнт - Сервер	18
HTML - Мова розмітки гіпертекстових документів	20
CSS - Каскадні таблиці стилів	24
Мова браузерів - Javascript	28
Специфіка фронтенду	30
Бекенд	36
Мова програмування Python	37
База даних MySQL	39
Веб-фреймворк Django	40
Репозиторій коду Git	42
Домашнє завдання	43
3. Проект: база даних для обліку студентів	45
Специфікації проекту	45
Чого ви навчитеся протягом проекту?	48
Що не входить у даний проект?	50
Домашнє завдання	51

ЗМІСТ

4. Робоче середовище та перший Django проект	53
Операційна система	53
Командна стрічка	55
Менеджер файлів	59
Редактор коду	61
Інсталюємо репозиторій коду Git	63
Інсталяція бази даних MySQL	65
Інсталяція Python	72
Інсталяція virtualenv та Django	83
Django Проект	86
Домашнє завдання	93
5. Верстаемо головну сторінку: лейаут та список студентів	95
Інструментарій	96
Основа HTML документа	98
Шапка та футер	105
Список студентів	114
Домашнє завдання	132
6. Динамізуємо головну сторінку	136
HTTP протокол	137
Що таке MVC?	141
Регулярні вирази	144
Створюємо Djangoapplікацію	147
Список студентів в Django	151
Статичні ресурси	173
Динамізуємо список студентів	180
Реалізуємо закладку Групи	184
Процесор контексту	195
Реорганізація в'юшок	201
Домашнє завдання	202
7. Розробляємо моделі Студента та Групи: моделі, поля, атрибути	204
База даних	204
ORM, Django моделі	210
Модель студента	215

ЗМІСТ

Оновлюємо список студентів	234
Навігація та сортування списку студентів	239
Модель групи	257
Фікстури та міграції	268
Домашнє завдання	273
8. Форми роботи із студентом та групою: Django форми, валідація	276
HTML Форми	277
Форма додавання студента	285
Форма контакту адміністратора	321
Форма редагування студента	344
Видалення студента	362
Кастомізація адміністративної частини Django	366
Домашнє завдання	374
9. Реалізуємо журнал відвідування: Javascript та AJAX в Django	376
Вступ до Javascript, AJAX, jQuery	377
Закладка Відвідування	387
Випадайка з групами	438
Віджет календаря для поля дати	452
Форма редагування студента в режимі AJAX	460
Домашнє завдання	482
10. Логування дій над студентами: сигнали в Django та Python логер	485
Теорія подій	486
Теорія логування	492
Ведемо журнал дій над студентами	512
Логуємо помилки при роботі форми контакту	521
Домашнє завдання	524
11. Перекладаємо інтерфейс проекту: інтернаціоналізація	527
Теорія інтернаціоналізації та локалізації	528
Готуємось до інтернаціоналізації	564
Перекладаємо шаблони	567
Перекладаємо Python код	584
Перекладаємо Javascript код	599
Активуємо мову під користувача	604

ЗМІСТ

Домашнє завдання	606
12. Доступ до аплікації: автентифікація користувачів	610
Теорія системи користувачів	611
Налаштовуємо середовище користувачів	618
Інтегруємо логін та реєстрацію	620
Розробляємо сторінку профіля	651
Інтегруємо Facebook логування	663
Домашнє завдання	676
13. Додатковий функціонал: мілдвара, команда, кастомний тег та фільтр	679
Django мілдвара	680
django-admin команда	692
Кастомні теги	699
Кастомний фільтр	710
Домашнє завдання	714
14. Автоматичні тести: покриваємо код аплікації	716
Що таке тести, коли їх писати та для чого?	716
Готуємо тестовий фреймворк	733
Тест утиліти	736
Тест форми та в'юшки	744
Тест моделі	766
Тест команди	769
Тест відправки листа	773
Тест обробника сигналу	776
Тест процесора контексту	782
Тест кастомного фільтру	784
Покриття коду тестами	788
Домашнє завдання	794

1. Вступ

Для кого і для чого дана книга?

Вітаю вас!

Якщо ви читаете даний текст, то це, швидше за все, означає одне з двох:

- ви вирішили опанувати веб-розробку і обрали першою мовою для себе - мову програмування Python;
- ви уже розбираєтесь у веб-розробці, але хочете освоїти ази створення веб-аплікацій саме з допомогою веб-фреймворка Django.

Завданням даної секції книги якраз і є розібратись у тому, для кого було написано дану книгу, чим вона відрізняється від інших книг, які ви читали до цього часу та взагалі чого вам варто від неї очікувати.

Завдання книги

Ви мабуть подумали: “А для чого ще одна книга з веб-програмування? Та ще й такого популярного фреймворка як Django. Адже уже існує маса книг як англійською так і російською мовами.” Тобто кожен бажаючий з легкістю може знайти необхідну інформацію щодо веб-розробки на даному фреймворку.

Я також так думав спочатку. Але згодом, працюючи із початківцями, зрозумів, що формат та підхід більшості існуючих книг є далеким від ідеального саме для людей, які ще не мають практики програмування. Навіть якщо книга базована на практичному проекті, то зазвичай формат подачі є швидше інформаційним та підходить людині, яка вже має хорошу практику програмування.

Із власного досвіду знаю, що початківцям недостатньо просто дати інформацію. Натомість, важливими чинниками успішності навчального матеріалу є:

- його практичне застосування;
- побудова усього матеріалу на прикладах;
- домашні завдання та заохочення до власних дій та покращення прикладів власними ідеями (тобто основний акцент на ініціативу студента та бажання вчитись розв'язувати проблеми використовуючи інформацію навчального матеріалу);
- наглядний матеріал, що безпосередньо показує процес вирішення проблем у живому непідготовленому режимі професійним програмістом (це як парне програмування);
- робота в команді собі подібних, де можна отримати як допомогу так і допомагати іншим;
- підтримка та мотивація, адже значно простіше долати перешкоди знаючи, що інші, такі як ти, також мають подібні проблеми на своєму шляху.

Якраз виходячи із вищезгаданих критеріїв успішного навчального матеріалу, саме для початківців я і сформував не просто практичний посібник з Django, а цілий пакет додаткових матеріалів та сервісів для підтримки та мотивації.

Таким чином можна виділити два основні завдання даної книги:

1. **для тих, хто ще немає достатньої практики програмування**, - показати як можна почати програмувати реальний проект. І не лише повторювати код, приклади та проект із книги, а ускладнювати їх власними ідеями та завданнями і самостійно їх реалізовувати. Таким чином набувати практики розв'язування наближених до реальних проблем під час створення веб-аплікації на Django.
2. **для тих, хто вже є веб-розробником**, - надати лише необхідний мінімум інформації для освоєння азів веб-розробки з Python та фреймворком Django.

Підсумовуючи усе вищезгадане: перша і найважливіша порада вам, якщо ви початківець і ще не маєте достатньо практики програмування:

Якщо ви не маєте достатньо часу, бажання, чи планів на те, щоб активно працювати із вашим редактором коду протягом даної книги, ускладнювати

проект власними ідеями, вчитись самостійно розв'язувати проблеми, що виникатимуть у вас при цьому, тоді прошу вас зупинитись відразу і даремно не гаяти власного часу. Ця книга винятково для тих, хто вже починаючи з 4-ї глави книги відкрисє свій улюблений редактор і почне регулярно виділяти час для кодування, читання чужого коду та розв'язування виникаючих проблем!

Якщо ж ви ще не визначились до кінця, що саме вас цікавить - чи вам потрібна веб-розробка, чи хочете мати справу з серверною стороною веб-сайтів, яку мову програмування обрати, що взагалі таке веб-розробка та програмування... - тоді пропоную спочатку ознайомитись із наступними матеріалами:

- Безкоштовний курс: “Програміст Початківець”¹
- Як знайти себе в IT?²
- 5 причин, які заважають вам стати програмістом уже завтра³
- Що таке веб-розробка та ваша можлива роль у ній?⁴
- Як обрати першу мову програмування та чому Python є ідеальним для початківців?⁵

Я впевнений, що дані матеріали дадуть відповіді на більшість ваших запитань. Якщо все ж маєте будь-які сумніви, тоді продовжуйте освоєння даної книги лише після ознайомлення з цими матеріалами.

...

Ну як? Переконались, що дана книга - це саме те, що вам потрібно?...

Тоді можемо рухатись далі. Ця і наступні 2 безкоштовні глави даної книги дадуть вам ще більше розуміння того, чим ми з вами займатимемось та куди нас це все приведе.

¹<http://www.vitaliyopoda.com/email-course-dummy-programmer/>

²<http://www.vitaliyopoda.com/it-webinar/>

³<http://www.vitaliyopoda.com/newbie-prog-webinar/>

⁴<http://www.vitaliyopoda.com/webdev-webinar/>

⁵<http://www.vitaliyopoda.com/webinar-python/>

Як працювати із даною книгою?

Перш за все хочу звернути вашу увагу на те, що дана книга йде у двох варіантах:

1. книга + код;
2. книга + код + закрита група підтримки + відео уроки + інтерв'ю з професійними програмістами + шпаргалки з технологій.

Якщо ви початківець, тоді вам потрібен саме другий варіант, який іде із всеможливими супорт матеріалами. Адже вам, як початківцю, не так важливо просто отримати інформацію, як отримати підтрику, мотивацію та детально показати (ідеально вживу) процес кодування та дебагу коду.

Щоб ефективно працювати із матеріалом даної книги, необхідний мінімум знань складає базове знання мови програмування Python включно з основами об'єктно-орієнтованого програмування (ООП). Якщо ви вважаєте, що даних знань вам бракує, тоді варто відкласти дану книгу; спершу підтягнути мову Python, а тоді вже повернутись і продовжити освоєння матеріалу даного підручника.

Ось кілька корисних джерел, де ви зможете швидко та якісно отримати базу мови програмування Python:

- англомовні Python курси на [CodeCademy⁶](http://www.codecademy.com/tracks/python), [Coursera⁷](https://www.coursera.org/course/pythonlearn), [Udemy⁸](https://www.udemy.com/python-for-beginners/) та [Udacity⁹](https://www.udacity.com/course/ud036);
- російськомовний відео курс “Ввід в Python”¹⁰ на Youtube;
- “Python 2: Курс Молодого Бійця”¹¹ у мене на блозі;
- набір посилань та рекомендацій із освоєння мови Python на [Habrahabr¹²](http://habrahabr.ru/post/150302/).

⁶<http://www.codecademy.com/tracks/python>

⁷<https://www.coursera.org/course/pythonlearn>

⁸<https://www.udemy.com/python-for-beginners/>

⁹<https://www.udacity.com/course/ud036>

¹⁰https://www.youtube.com/playlist?list=PLEl2mW_X5hhkgW_e7B_ukSwPm3Q28eSyt

¹¹<http://www.vitaliyopodoba.com/tutorials/python2-beginners-course/>

¹²<http://habrahabr.ru/post/150302/>

На сторінках цієї книги я даватиму посилання на зовнішні курси та навчальні матеріали у випадку, якщо та чи інша глава вимагатиме певних азів технологій або мови (як от HTML, CSS чи Javascript).

Завдання книги - надати вам інформацію та приклади коду із поясненням азів веб-програмування з Django. Кожна глава закінчується домашнім завданням. Завдання починаються від найпростіших і закінчуються великими ідеями з покращення коду та функціоналу побудованого у даній главі. Таким чином маєте ідеї для подальшої практики.

Зазвичай, особливо на перших порах, у вас виникатиме маса проблем та запитань в процесі виконання домашніх завдань. І чим більше ви відходите від книжкових прикладів, тим більше виникатиме труднощів.

Саме для того, щоб не здаватись і доводити домашні завдання до кінця, до книги (Рекомендований пакет матеріалів) додається закрита група. Там можна обговорити те чи інше домашнє завдання, поставити конкретне технічне запитання та отримати на нього відповідь. У групі є як професійні програмісти, які добре розбираються у всіх аспектах веб-програмування, так і такі ж початківці, як ви. Така закрита тусовка значно збільшить ваші шанси для набуття практики програмування.

Із книгою також йде відео курс із сесіями програмування. Курс містить відео запис розробки того ж проекту, який описано в книзі, але форма подачі кардинально інша. Курс спеціально створено, як то кажуть, з першої спроби. Без спеціальної підготовки. Для чого? А для того, щоб ви, переглядаючи відео, могли бачити справжню живу сесію кодування професійного програміста, у якого в процесі роботи виникають справжні проблеми, які він розв'язує. Саме так. Відео містить усі проблемні місця включно із дебагом коду, брейкпоінтами, гуглінням, ковирянням коду та читанням коду фреймворку Django. Завдання відео курсу - навчити вас не просто робити все згідно з інструкцією, але й дати інструментарій та підходи для самостійного розв'язання ваших власних проблем під час програмування.

Адже програміст - це не той, хто знає мову програмування, а вміє скористатись нею для розв'язання реальних проблем та задач. Таким чином, відео курс, який

іде із повним пакетом книги, дасть вам відчуття парного програмування з професійним програмістом.

Даний відео курс рекомендую опрацьовувати паралельно з книгою, саме під час роботи над домашніми завданнями.

Також із книгою йде набір шпаргалок із Python, HTML, CSS та Git. Усі вони допоможуть вам сфокусуватись на програмуванні і не гаяти часу в пошуках тієї чи іншої функції в мові Python, тегу в HTML чи команди в репозиторії коду Git. З часом ви запам'ятаєте найбільш популярні функції, теги, правила, команди, але на початках дані підказки допоможуть вам добряче зекономити час і натомість проводити його над розв'язуванням безпосередніх завдань.

І як додатковий бонус до повного пакету книги додається кілька відео інтерв'ю із програмістами: як професіоналами, так і з джуніками (тими, що вже працюють). Рекомендую переглянути їх із самого початку, а також звертатись до них щоразу, коли рівень мотивації знижуватиметься, а рівень розчарування підвищуватиметься. В даних інтерв'ю ви знайдете історії людей, які вже вміють те, чого ви хочете навчитись. Ви отримаєте їхні поради щодо навчання програмуванню, пошуку роботи, ну і звичайно мотивацію та розуміння того, що не так вже й все складно.

Отже, підведемо підсумок щодо роботи із даною книгою:

- спочатку переглядаємо відео інтерв'ю із програмістами, мотивуємось, складаємо власний план досягнення цілей;
- далі налаштовуємось на довготривалу та плідну роботу, без халяви;
- далі читаємо главу книги, пробуємо приклади із прочитаної глави;
- якщо щось не виходить, як описано, - звертаємось за допомогою у закриту групу підтримки (спочатку перевірте чи вже немає відповіді на ваше запитання у групі, на більшість запитань початківця зазвичай існують відповіді від попередників, адже у всіх спільні проблеми);
- далі практика: працюєте над домашніми завданнями;
- під час роботи над домашніми завданнями виникатиме маса проблем і запитань, для вирішення яких знову звертаєтесь у закриту групу;
- також на етапі роботи із домашніми завданнями варто переглядати відео курс, з якого черпати практику вирішення реальних задач, користування інструментами для відлагодження коду;

- в процесі кодування користується шпаргалками.

Цю книгу я постійно вдосконалював і переписував під час роботи зі студентами, яких я менторив в напрямку веб-програмування на Django. Адже студенти давали мені живий фідбек в якості запитань та скарг стосовно тих речей, які вдавалися найважче. Таким чином я змінював пояснювальні тексти, а також додавав речі, які на перший погляд були для мене очевидними, але насправді викликали масу запитань. Саме завдяки студентам, я вважаю, мені вдалось дохідливо організувати книгу та додаткові матеріали, що врешті дозволить вам значно легше опановувати веб-програмування та набувати практику.

На завершення даної секції ще хочу повідомити вам, що дана книга не буде разовою роботою із моєї сторони, а натомість постійно вдосконалюватиметься новим та кращим матеріалом, відповідно до фідбеку читачів та нових версій Python та Django. Я планую постійно і на регулярній основі підтримувати матеріал даної книги в хорошій формі та в майбутньому релізити нові версії. Усі інші мої продукти будуть створюватись на базі даної книги таким чином, щоб надати допомогу початківцю на усіх його етапах від навчання азів програмування і аж до пошуку першої роботи.

Форматування у книзі

Як ви уже мабуть зауважили, книгу я старався писати простою живою мовою без зайного офіціозу. Приблизно такою мовою, якою я зазвичай пишу статті на своєму блозі. Адже вважаю, що таким чином можна краще надати практичний матеріал, передати власний досвід, організувати та змотивувати читача до дій, ніж пишучи сухою офіційною мовою книги. Час до часу можуть проскачувати сленгові слова, суржик, русизми, англіцизми і т.д. Я не намагався їх зменшити в книзі, натомість хочу, щоб книга звучала автентично та передавала настрій автора і давала вам відчуття реального персонального ментора з усіма його нюансами :-).

Окрім звичайного тексту, в даній книзі також використовується форматування блоків коду. Зазвичай, це спеціально підсвічений блок коду. Кожна мова матиме свою підсвітку. Ось приклад Python та HTML коду:

Приклад Python коду

```
1 class BaseExamFormView(object):
2     def post(self, request, *args, **kwargs):
3         # handle cancel button
4         if request.POST.get('cancel_button'):
5             return HttpResponseRedirect(reverse('exams_list') +
6                 '?status=Зміни скасовано.')
7         else:
8             return super(BaseExamFormView, self).post(
9                 request, *args, **kwargs)
```

Приклад HTML коду

```
1 <html>
2     <head><title>Заголовок Сторінки</title>
3     <body>
4         <h1>Приклад Сторінки</h1>
5         <p>Підсвітка Рулить!</p>
6     </body>
7 </html>
```

Здебільшого книга містить лише практичний матеріал і лише той, що стосується проекту, який розроблятимемо на сторінках даного посібника.

У тих місцях, де виникатиме потреба у невеликій додатковій теоретичній інформації, зауваженнях, практичних порадах чи нотатках від автора, ми використовуватимемо ось такі блоки тексту, огорнуті в рамку.

Які технології розглянемо у даній книзі?

На сторінках даної книги ми створимо наближений до реального веб-проект, в процесі якого працюватимемо із наступними технологіями:

- мова програмування [Python¹³](http://python.org/);
- веб-фреймворк [Django¹⁴](https://www.djangoproject.com/);
- бази даних [sqlite¹⁵](http://www.sqlite.org/) та [MySQL¹⁶](http://www.mysql.com/);
- мова розмітки [HTML¹⁷](http://www.w3.org/TR/html/);
- каскадні таблиці стилів [CSS¹⁸](http://www.w3.org/TR/CSS/);
- мова програмування [Javascript¹⁹](http://uk.wikipedia.org/wiki/JavaScript);
- Javascript бібліотека [jQuery²⁰](http://jquery.com/);
- HTML/CSS фреймворк [Twitter Bootstrap²¹](http://getbootstrap.com/2.3.2/);
- репозиторій коду [Git²²](http://git-scm.com/).

У наступних главах ми детальніше поговоримо про кожну із вищезгаданих технологій.

Тут лише зауважу, що основний акцент даної книги є на серверній розробці веб-сайту. Ми розроблятимемо також і клієнтську сторону веб-аплікації, але для цього використовуватимемо бібліотеку Twitter Bootstrap, що полегшить наше завдання та дозволить менше занурюватись у верстку, а більше фокусуватись на Django фреймворку.

Організовуємось!

Як ви вже зрозуміли, дана книга - це не просто інформація з веб-програмування, а спроба надати справжню підтримку та підвищити шанси студента на успіх.

Початківець на шляху до успіху зустрічається з двома величезними проблемами, з якими варто навчитись працювати:

¹³<http://python.org/>

¹⁴<https://www.djangoproject.com/>

¹⁵<http://www.sqlite.org/>

¹⁶<http://www.mysql.com/>

¹⁷<http://www.w3.org/TR/html/>

¹⁸<http://www.w3.org/TR/CSS/>

¹⁹<http://uk.wikipedia.org/wiki/JavaScript>

²⁰<http://jquery.com/>

²¹<http://getbootstrap.com/2.3.2/>

²²<http://git-scm.com/>

1. організація та самодисципліна;
2. мотивація та віра у власні сили.

Саме тому, окрім матеріалів по програмуванню, в дану книгу я також додав поради та рекомендації із самоорганізації. Вони допоможуть вам освоїти матеріал книги, отримати практику програмування, а не просто прочитати і забути.

Тому для початку давайте розберемо кілька важливих моментів щодо організації свого часу, щоб збільшити наші шанси на успіх.

Сила “Вже і Зараз”

Якщо ви ще цього не зрозуміли, то знайте: те, що ми робимо прямо тут і зараз безпосередньо впливає на наше майбутнє. Кожна наша думка, рішення і дія формують наше майбутнє, допомагають або заважають нам досягати цілей та формувати вдалі плани.

Саме тому так важливо перестати відкладати на завтра ті речі, які можна спробувати уже сьогодні, через годину, або ще краще прямо зараз!

Виходячи із даного принципу, рекомендую вам, особливо якщо ви ще початківець у програмуванні, виконувати приклади та домашні завдання з даної книги “не відходячи від каси”. Не відкладайте на завтра, адже завтра будуть нові ще цікавіші завдання.

Якщо хочете знати, чому ми усі відкладаємо найважливіші справи у нашому житті на завтра, а також те, як з цим бути, тоді почитайте мою статтю на дану тему. Вона дасть вам гарне розуміння ситуації, а з ним прийде і бажання щось змінити: **“Як початківцю не відкладати на завтра той код, що можна закодити уже сьогодні!”²³**.

Правильний розпорядок дня

Наш мозок працює згідно 4-ох годинних фаз. Перші 4 години вашого дня він зазвичай є найактивнішим, має величезний потенціал та енергію до

²³<http://www.vitaliyopoda.com/2014/09/start-coding-today/>

вирішення найважчих проблем дня. Потім настає 4 години обіднього часу, коли йому варто відпочити. І далі, після обіду, знову 4 години, коли він вже не такий свіжий, як на початку дня, але ще у гарному стані для продовження менш інтенсивної розумової праці. Як от, наприклад, навчання.

Звісно, у кожного з нас є свій розпорядок. Хтось любить вставати рано-вранці, а хтось довше поваляється в ліжку. Комусь програмується значно краще вночі, а комусь після обіду. Це все справа звички.

В будь-якому випадку рекомендую вам, по-перше, виділити фіксований проміжок часу у вашому дні, коли ви будете займатись практикою програмування. В ідеалі хоча б 2 години, ну або хоча б 1 годину. Але тоді мати таких проміжків хоча б два на один день.

По-друге, так само виділити час на навчання. Це може бути прочитання книги, проходження курсу, перегляд відео туторіалів і т.д.

По-третє, переконатись, що проміжок практики програмування завжди йде перед періодом навчання. Адже навчання не потребує такої кількості енергії та свіжості вашої думки, як програмування. Саме тому, першою у вашому денному графіку має бути практика програмування, а вже потім - навчання програмуванню та теорія.

І взагалі, значно краще щодня приділяти невелику кількість часу програмуванню, ніж один раз на тиждень по кілька годин за присіст. Наша підсвідомість працює так, що їй краще запам'ятовувати і обробляти інформацію регулярно невеликими порціями, таким чином перебуваючи постійно в атмосфері предмету, який ви освоюєте.

У даній статті я описав, як виглядає мій робочий день, і пояснив чому найважчі справи я роблю із самого початку. Рекомендую взяти собі на озброєння кілька ідей: “Робочий день програміста або як не проживати все життя за компом”²⁴.

Сила звички

Буде взагалі ідеально, якщо вищенаведені рекомендації із розпорядку вашого дня ви зможете ввести у своє життя на регулярній основі. На рівень звички.

²⁴<http://www.vitaliyopoda.com/2014/08/programmer-day/>

Мотивація, бажання, мрії, зобов'язання - все це з часом проходить і перестає працювати. Але якщо до моменту, коли це все пройде, ви зможете виробити у себе звичку кожного дня хоча б годину програмувати, то це буде запорука успіху для вас як початківця у програмуванні.

Ще донедавна я і не уявляв, що звичка може бути таким потужним інструментом і ключем до всього! Хочете побудувати успішний бізнес? Випрацюйте правильні звички. Хочете поїхати у кругосвітню подорож? Випрацюйте правильні звички. Хочете навчитись професійно програмувати? Випрацюйте правильні звички.

Лише завдяки звичці я не зупиняюсь писати статті у своєму блозі. Лише завдяки звичці я написав дану книгу. Тому я впевнений, що, якщо ви виробите у собі правильні звички, вони гарантуватимуть вам успіх у вивченні програмування та освоєння фреймворку Django.

Як виробляти звички? Ось невеличкий огляд того, як я це робив: “[Сила Звички](#) або Як Досягати Результату²⁵”.

Тайм менеджмент

На завершення організаційної секції хочу згадати, що існує маса технік та підходів з управління своїм часом. Яка з них підійде саме вам? Не знаю. Потрібно самому багато різних речей спробувати і зрозуміти, що найкраще працює для вас.

Коли я багато програмував протягом дня, то успішно використовував [Техніку Помодоро²⁶](#). Можливо вона допоможе і вам.

Із власних спостережень я зрозумів, що розподіл свого часу на менші частини допомагає мені краще його використовувати і менше гаяти на непотрібні речі. До прикладу, якщо кожні 30 хвилин роблю перерву на 5-ти хвилинний відпочинок, тоді встигаю більше, ніж коли працюю, не встаючи 2 години. За 30 хвилин я просто не встигаю відволікатись на непотрібні речі.

А загалом, треба навчитись домовлятись із собою. Кожен раз, коли займаєтесь не надто важливою справою, запитайте себе: “А чи це приведе мене через

²⁵<http://www.vitaliyopoda.com/2014/04/the-power-of-habit/>

²⁶<http://www.vitaliyopoda.com/2014/04/pomodoro-accomplish-more-in-25-minutes/>

тиждень (місяць, рік, 10 років) до моєї цілі?”. Мені допомагає.

Експериментуйте і шукайте свої підходи для самоорганізації та самодисципліни, адже без них далеко не зайдеш.

Мотивуємось!

У мене вже неодноразово були ситуації, коли студент на персональному менторстві опускав руки, розчаровувався і готовий був здатись буквально у перший тиждень співпраці.

Я, як ніхто інший, тепер знаю, що мотивація, віра в себе, порада з боку - є одними з найпотужніших інструментів у допомозі початківцю освоїти програмування.

Що ж стойть на шляху початківця, що може швидко обламати його крила у перші тижні практики?

Несвідомий код

Мабуть найбільшою проблемою людини, яка починає розбирати багато технологій, мов і напрямків одразу (а так воно у вебі часто буває), є розчарування через те, що багато речей на практиці якщо і вдаються, то вдаються несвідомо. Без повного розуміння усієї картини.

Дядько Google, “copy-paste” (скопіював-вставив), “метод наукового тику”, випадкові спроби та експерименти - ось це все з чого я сам починав програмувати. І це незважаючи на те, що я перед тим прочитав з 5-7 книг по усіх потрібних мовах та технологіях.

Якщо ви лише починаєте набиратись практики програмування, тоді, впевнений, у вас буде подібна ситуація. Адже ніхто не починає ходити в дитинстві допоки 1000 разів не впаде.

Тому не бійтесь несвідомого коду. Просто сідайте і пишіть. Чим більше напишете, тим швидше перейдете на рівень свідомого коду.

Ось рівні коду початківця:

1. написати будь-який код;
2. зробити цей код таким, щоб запускався, тобто без синтаксичних помилок;
3. оновити код так, щоб він працював;
4. оновити код так, щоб він працював правильно;
5. заглянути ще раз в середину коду і тепер зробити, щоб він ще й виглядав гарно згідно з кращими практиками мови та технологій;
6. дописати тести та документацію (якщо звичайно має зміст у даній ситуації);
7. вернутись до свого коду за деякий час і виявити у собі бажання переписати код ще кращим чином. Якщо бажання з'явилось, значить за цей час людина прогреснула.

Тому, аби перейти на рівень 7, треба якомога швидше почати рівень 1. Крапка. І без розчарувань та ідеалів щодо результатів від прочитання теоретичних книг.

Переломний момент

З часом рівень “несвідомого коду” знижується і чим далі, тим з більшою швидкістю. Варта лише не здатись на початку при перших проблемах, а далі кожен наступний виклик буде долатись вами простіше і простіше.

Кажуть, що для того, щоб стати експертом у будь-якій справі, потрібно провести як мінімум 10.000 годин свідомої практики.

Так от, я думаю, що для програміста-початківця, щоб дійти до переломного моменту після якого можна **самостійно** (хоч інколи і повільно) вирішувати більшість проблем у веб-розробці з Django, потрібно в середньому 6-8 тижнів регулярної щоденної практики. Мінімум 2-3 години програмування на день. Це рівень, коли вам вже не потрібен наставник для вирішення кожної проблеми, а більшість питань ви з'ясовуєте та відпрацьовуєте самостійно.

Після 6-8 тижнів ви не станете професіоналом, але протягом них початківець зазвичай переходить із першого рівня написання коду до написання коду, що працює правильно. І, що дуже важливо, - самостійно, без допомоги ззовні.

Тому перші кілька тижнів є особливо важкими. На них варто відповідно налаштуватись, змотивуватись і зрозуміти, що дуже багато речей буде незрозуміло. Але тим не менше, доведеться просто сідати і писати, ковиряти та читати чужий код.

Просто скажіть собі: “це нормальну, так має бути, це не я невдаха, це усі так починають!”. Зрозумійте, що більшість професійних програмістів саме так і починали.

Вища мета

Кожного разу, коли:

- я хочу відкласти важливу некомфортну справу на пізніше,
- маю великі несподівані проблеми на своєму шляху,
- не впевнений у правильному виборі,
- просто лінуюсь,

я стараюсь зупинитись і згадати про свою вищу мету, про план та цілі, які я розробив раніше. Зазвичай, кожна поточна проблема вирішується легше, якщо ви згадуєте те, що вас чекає коли успішно подолаєте даний етап.

Вища мета поповнює вашу енергію та додає мотивації у кожній складній ситуації. Головне не забувати про неї і регулярно піднімати голову над буденними проблемами.

Але, щоб цей метод досягнення цілей працював, переконайтесь, що ваша мета та цілі є дійсно такими, від яких у вас захоплює дух. Такими, які посправжньому мотивують вас та з легкістю піднімають вас кожного ранку з ліжка.

Ви мабуть зауважили, що на моєму блозі близько половини усіх статей мають мотиваційний та підбадьорюючий характер. Я дуже стараюсь, щоб дозволити людині глянути на велику гору проблем, як на маленький посильний горбик.

Адже в протилежному випадку, більшість людей навіть не почне сходження на неї. Ось лише декілька із цих статей, читайте, мотивуйтесь і повертайтесь до них у важкі хвилини:

- Мої Спостереження про викладання та навчання програмуванню²⁷,
- Чотири Кроки до Гугла без Університетського Ступеня²⁸,
- 5 способів, щоб стати найкращим у своїй справі²⁹,
- Мої поради усім новим програмістам або як мотивуватись?³⁰.

Домашнє завдання

Ви можете здивуватись, чому вступна глава займає так багато сторінок і чому він стільки “лиє води”? Коли ми вже нарешті перейдемо до діла і почнемо програмувати?

На це я вам скажу, що дана глава насправді є однією із найважливіших у цій книзі. Питання підтримки, самоорганізації та мотивації є найважливішими для початківців у програмуванні. Саме тому я ще не один раз буду заряджати сторінки даної книги “тайм менеджмент” фішками, “мотивашками” та іншими речима, які не мають прямого відношення до програмування.

Але, якщо ви супер-організований, оптимістичний та вмотивований супермен, тоді надалі можете сміливо пропускати такі ліричні відступи.

Усім іншим щиро раджу:

- переглянути свій робочий день;
- переглянути своє питання: “Для чого мені усе це?”;
- поставити далекоглядні високі цілі, які вас будуть по-справжньому мотивувати;
- виділити собі години для роботи із книгою та години для самого програмування;

²⁷<http://www.vitaliyopoda.com/2013/08/my-observations-teaching-and-learning-programming/>

²⁸<http://www.vitaliyopoda.com/2014/04/four-steps-to-google-without-degree/>

²⁹<http://www.vitaliyopoda.com/2014/06/how-to-be-best/>

³⁰<http://www.vitaliyopoda.com/2014/08/what-i-tell-all-new-programmers/>

- в нашій закритій групі підтримки написати про себе і познайомитись із тусовкою.

Якщо все зробите правильно, тоді вважайте, що 70% успіху у вас в кишенні!

В наступній главі ми з вами оглянемо ази веб-розробки. Розглянемо, що таке створення веб-сайту, які технології і мови задіяні у процесі його створення і гарно підготуємося до нашого веб-проекту.

2. Що таке веб-розробка?

У цій главі поговоримо про те, що таке інтернет для нас і як він працює. Які технології використовуються при створенні нового веб-сайту та те, як взаємодіють сервер, що обслуговує веб-аплікацію та веб-переглядач (або ще як його називають веб-браузер), який дає можливість користуватись даною аплікацією.

Думаю ви уже можете дещо знати про веб-розробку та суміжні технології, але щоб усі читачі та студенти починали маючи один і той самий необхідний мінімум теоретичних знань, спільний старт, пропоную ще раз коротко оглянути основи веб-розробки та те, з чого складається *інтернет*.

Комунікація Клієнт - Сервер

Отже, що таке інтернет для нас як користувачів? Це набір різних порталів, до яких ми звикли, користуємося щоденно, шукаємо інформацію, переглядаємо відео, слухаємо музику, комунікуємо з друзями, читаємо новини і тому подібне.

Для того, щоб ми могли користуватись інтернетом нам потрібен браузер - програма, яка дозволяє переглядати та користуватись веб-сайтами. У наш час маємо кілька найпопулярніших браузерів, серед яких є Chrome, Firefox, Internet Explorer, Safari та Opera.

Програма браузер дозволяє нам взаємодіяти із вебсайтом, дані якого (всеможливі файли та сторінки) лежать на віддаленому сервері.

Для того, щоб взаємодіяти із сервером існує спеціальний формат адрес під назвою URL (Universal Resource Locator - Універсальний Локатор Ресурсів). Саме ця адреса дозволяє визначити, на якому віддаленому сервері знаходитьться сторінка, яку нам потрібно. Також адреса містить шлях до цієї сторінки чи файлу в межах того ж таки віддаленого сервера.

Ось приклад URL адреси сторінки продажу даної книги:

<http://www.vitaliyopodoba.com/books/django-for-beginners>

Частина *www.vitaliyopodoba.com* вказує на віддалений сервер, де знаходиться мій вебсайт-блог. А частина адреси “/books/django-for-beginners” вказує, де саме знаходиться сторінка з книгою в межах того віддаленого сервера.

Також адреса може містити додаткові параметри, які йдуть одразу за знаком запитання наприкінці:

<http://www.vitaliyopodoba.com/books/django-for-beginners/buy/?package=recommended>

Але детальніше про формат адрес ми ще поговоримо під час подальшої розробки нашого з вами практичного проекту.

Зазвичай по дорозі між веб браузером та віддаленим сервером знаходиться велика кількість інших серверів, що працюють в інтернеті. І кожен наш запит на сервер проходить усі ці віддалені сервера. Саме завдяки їм ми маємо доступ до кінцевого сервера, де знаходиться потрібний нам сайт.

Інформація, якою обмінюються веб переглядач та віддалений сервер зазвичай представлена у форматі [HTML³¹](#). Про нього ми трохи детальніше поговоримо у наступних секціях даної глави. Формат HTML описує вміст сторінки, яку показує веб браузер.

Оскільки ми маємо кілька серверів у ланцюжку, що з’єднує наш веб переглядач із кінцевим віддаленим сервером, усі вони повинні спілкуватись між собою у певному наперед узгодженному стандарті. Ці стандарти комунікації серверів в інтернеті описуються такими словами як:

- [TCP/IP³²](#) - протокол, що пояснює серверам як вони повинні передавати інформацію між собою на низькому рівні;
- [HTTP³³](#) - протокол, який пояснює браузеру як саме робити запит за документом із сервера, а серверу, у свою чергу, пояснює як готовувати відповідь браузеру. Його ви мали бачити, як мінімум, у більшості адрес веб-сайтів, адже вони починаються із “<http://>” стрічки.

³¹<http://uk.wikipedia.org/wiki/HTML>

³²<http://uk.wikipedia.org/wiki/TCP/IP>

³³<https://uk.wikipedia.org/wiki/HTTP>

Таким чином браузер може робити *запит* на сервер за певною URL адресою у потрібному форматі згідно HTTP протоколу, а сервер обробляє даний запит, готує *відповідь* також у наперед визначеному форматі (згідно HTTP протоколу).

Оскільки браузер і сервер знають протокол HTTP і знають як спілкуватись між собою, вони можуть обмінюватись документами, а ми, відповідно, бачити результат цього обміну у вигляді веб-сайтів та веб-сторінок.

Відповідно далі постає питання, а як працюють самі сторінки?

HTML - Мова розмітки гіпертекстових документів

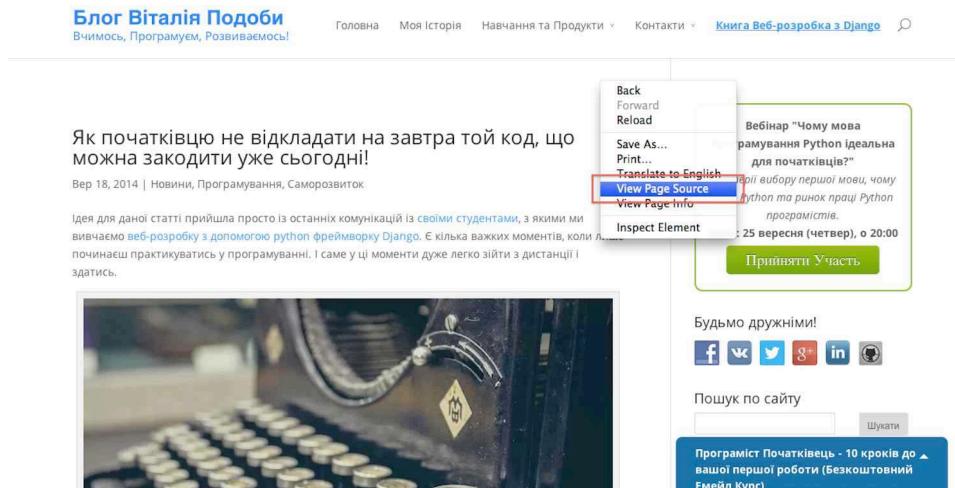
Більшість сторінок в інтернеті побудована саме з допомогою мови HTML (HyperText Markup Language - Мова Розмітки Гіпертекстових Документів).

З Вікіпедії: Гіпертекст (англ. Hypertext) – текст для перегляду на комп’ютері, який містить зв’язки з іншими документами (“гіперзв’язки” чи “гіперпосилання”); читач має змогу перейти до пов’язаних документів безпосередньо з вихідного (первінного) тексту, активізувавши посилання. Найпопулярнішим зразком гіпертексту є World Wide Web, у якому веб-оглядач переміщує користувача з одного документу на інший, щойно той «натисне» на гіперпосилання.

Тобто наші старі добре посилання, або ще як ми їх просто називаємо “лінки”, і є тою причиною, чому ми веб сторінки також називаємо “Гіпертекстом”. Саме це слово є складовою абревіатури HTML.

Таким чином мова HTML є основою основ усіх веб сайтів, а значить і веб-розробки. З допомогою цієї мови ми можемо описати структуру сторінки, її ієархію (так, HTML документ є ієархічною структурою вкладених HTML тегів), вставити параграфи тексту та заголовки, списки і таблиці, зображення, відео та звук, реалізувати форми.

Якщо зайти на будь-який сайт і клацнути там правою клавішою миші на вільному від тексту і зображень місці на сторінці, отримаємо контекстне меню, у якому знайдемо елемент “View Page Source” або “Перегляд HTML Коду”, або щось подібне (взалежності від вашої встановленої мови та веб переглядача).



Як переглянути HTML код сторінки

Вибрали його вам відкриється нове вікно переглядача із HTML кодом даної сторінки. Приблизно так всередині виглядають більшість веб сторінок в інтернеті:

```
<html lang="uk">
<!--[if !endif]-->
<head>
    <meta charset="UTF-8" />
    <title></title>
    <meta name="description" content="" /> <meta name="keywords" content="" /> <script type='text/javascript'>window.mod_pagespeed_start = Number(new Date());</script><link rel="canonical" href="http://www.vitaliyopodoba.com"/>
    <link rel="pingback" href="http://www.vitaliyopodoba.com/xmlrpc.php"/>
    <!--[if lt IE 9]>
    <script src="http://www.vitaliyopodoba.com/wp-content/themes/Divi/js/html5.js" type="text/javascript"></script>
    <![endif]-->
    <script type="text/javascript">
        document.documentElement.className = 'js';
    </script>
    <link rel="alternate" type="application/rss+xml" title="Віталій Подоба &raquo; Стрічка" href="http://www.vitaliyopodoba.com/feed/" />
    <link rel="alternate" type="application/rss+xml" title="Віталій Подоба &raquo; Команди та коментарі" href="http://www.vitaliyopodoba.com/comments/feed/" />
    <meta content="WP Divi v.1.0.0" name="generator" /><link rel="stylesheet" id="mailchimpSF_main_css-css" href="http://www.vitaliyopodoba.com/h/2.3.2.pagespeed.ce.4H7wTq1?T.css" type="text/css" media="all" />
    <!--[if IE]>
    <link rel="stylesheet" id='mailchimpSF_ie_css-css' href="http://www.vitaliyopodoba.com/wp-content/plugins/mailchimp/css/ie.css?ver=3.9.2" type="text/css" media="all" />
    <![endif]-->
    <link rel="stylesheet" id='crayon-css' href="http://1.ps.googleusercontent.com/h/www.vitaliyopodoba.com/wp-content/plugins/crayon-syntax-highlighter/css/min/crayon.min.css,qver=2.6.5.pagespeed.ce.yk5ohh0XR.css" type="text/css" media="all" />
```

Кусок HTML коду сторінки на моєму блозі vitaliyopodoba.com

Якщо у спрощеному варіанті, то структура HTML документу виглядає приблизно отак:

Приклад HTML документу

```
1 <html>
2   <head>
3     <meta charset="UTF-8" />
4     <title>Приклад HTML Сторінки</title>
5     <meta name="description" value="Мета опис сторінки" />
6   </head>
7   <body>
8     <h1>Заголовок HTML сторінки</h1>
9     <p id="paragraph">Приклад параграфа тексту</p>
10    <br />
11    <a class="my-link" href="http://google.com">Посилання</a>
12  </body>
13 </html>
```

Якщо збережете код вище наведеного прикладу у файл у себе на комп'ютері та відкриєте у себе в браузері, тоді отримаєте щось подібне на оце:

Заголовок HTML сторінки

Приклад параграфа тексту

[Посилання](#)

Приклад HTML сторінки у браузері

Як бачите основу HTML складають *теги*. Це елементи огорнуті в кутові дужки. Якщо глянете на 4-ий рядочок прикладу вище, то побачите тег “*title*”. Він складається із відкриваючого елементу “*<title>*”, вмісту “Приклад HTML Сторінки” та закриваючого елементу “*</title>*”. Є теги, які не мають вмісту, а також є теги, які не мають закриваючого елемента. Приклад можете бачити у вище наведеному коді на 10-му рядочку: тег переводу рядка “*
*”.

Також теги можуть мати властивості, так звані *атрибути*. 5-ий рядок коду у прикладі вище містить тег “*meta*” з двома атрибутами “*name*” (ім’я) та “*value*”

(значення). Кожен із атрибутів має ім'я та значення, що іде одразу після ім'я та знаку “=”. Кожен тег має власний набір атрибутів притаманний саме йому.

Структура документу зазвичай містить такі обов'язкові елементи як

- *html*: кореневий елемент будь-якого HTML документу;
- *head*: шапка документу; містить метадані сторінки та під'єднання інших ресурсів до сторінки (такі як javascript та css файли);
- *body*: тіло документу; містить теги, що є видимою частиною веб-сторінки.

Існує велика кількість HTML тегів, але найбільш поширеніх, які ми будемо з вами використовувати найчастіше, є лише в межах 20-ти.

В шпаргалці по HTML та CSS, яка йде із рекомендованим набором до даної книги, ви можете ознайомитись із найбільш поширеними елементами та їх застосуванням.

Детальніше із синтаксисом мови HTML можете ознайомитись на [сторінці вікіпедії](#)³⁴. А щоб отримати мінімум практики та розуміння як вона працює в браузері, рекомендую пройти один-два базових онлайн курси. Далі в книзі я буду пояснювати нюанси мови HTML рівно на стільки, на скільки це буде нам необхідно для реалізації нашого проекту.

На завершення даної секції хочу зауважити, що при освоюванні мови HTML варто також звернути увагу на такі поняття як [семантичний HTML](#)³⁵, [доступність у вебі](#)³⁶ та [мікро-формати](#)³⁷. Таким чином в подальшій своїй практиці ви верстатимете сторінку не лише так, щоб вона виглядала як в оригінальному дизайні, але й правильно з точки зору внутрішнього коду.

³⁴<http://uk.wikipedia.org/wiki/HTML>

³⁵<http://webgyry.info/semantics-html>

³⁶<http://bit.ly/vpwebaccessibility>

³⁷<http://bit.ly/vphmlmicroformats>

HTML чи HTML5?

Це є уже традиційним запитанням початківців щодо того, яку версію мови HTML вивчати: HTML 4.01 чи HTML 5?

HTML 5 - це наступна версія мови HTML із значними покращеннями в сторону семантичного (змістового) вебу та новими мультимедійними можливостями. Під семантикою розуміємо надання тегам не лише функції для представлення даних, але й і для пояснення цілі цих даних.

HTML 5 є свого роду уніфікацією XML, HTML 4.01, XHTML. Робота над даним стандартном почалась ще у 2004 році і триває і досі і ще буде тривати далеко не один рік. Різні веб браузери підтримують уже цілий ряд HTML 5 функціоналу, але є відмінності між їхніми реалізаціями.

Тому, якщо давати коротку відповідь з якої версії HTML мови починати, то звичайно починайте із поточної HTML 4.01. Куди б HTML 5 в майбутньому не повернула, HTML 4.01 все одно буде хорошою базою для неї ще довгий час.

Детальніше про різноманітні варіанти стандарту HTML, такі як [XHTML³⁸](#), [HTML5³⁹](#), і т.д. можете ознайомитись знову ж таки на [сторінці вікіпедії⁴⁰](#).

CSS - Каскадні таблиці стилів

З часом прості веб-сторінки із одноманітним нецікавим текстом усім набридли, тому почали придумувати та додавати стилі до HTML елементів. Спочатку це було у вигляді нових атрибутів напряму в HTML теги, а також через табличні стилі для створення потрібного лейауту сторінки.

Згодом винайшли так звані каскадні таблиці стилів - нову мову розмітки для описування стилів для елементів на сторінці - CSS. Каскадні тому, що вони дозволяють визначати стилі елементів на різних рівнях і унаслідувати стилі від вищих в ієрархії елементів.

³⁸<http://en.wikipedia.org/wiki/XHTML>

³⁹<http://en.wikipedia.org/wiki/HTML5>

⁴⁰http://en.wikipedia.org/wiki/HTML#WhatWG_HTML_versus_HTML5

CSS давав надзвичайні переваги, адже його можна було описувати в окремому файлику, а тоді під'єднувати до документа веб-сторінки. Таким чином змінивши кілька стрічок в CSS файлі можна було одним махом оновити верстку на усьому сайті.

Давайте продовжимо із нашого прикладу із секції HTML:

Приклад HTML документу

```
1 <html>
2   <head>
3     <meta charset="UTF-8" />
4     <title>Приклад HTML Сторінки</title>
5     <meta name="description" value="Мета опис сторінки" />
6   </head>
7   <body>
8     <h1>Заголовок HTML сторінки</h1>
9     <p id="paragraph">Приклад параграфа тексту</p>
10    <br />
11    <a class="my-link" href="http://google.com">Посилання</a>
12  </body>
13 </html>
```

Спробуємо зробити розмір заголовка трохи меншим, розмір тексту параграфа трохи меншим і курсивом, а посилання зробимо білим кольором на чорному фоні та ще й поставимо його в правому верхньому куті сторінки.

Ось CSS код, що для нас все це зробить:

Приклад CSS стилів

```
1 h1 {
2   font-size: 16px;
3 }
4 p#paragraph {
5   font-size: 14px;
6 }
7 a.my-link {
8   position: absolute;
9   top: 10px;
10  right: 20px;
11  display: block;
12  width: 200px;
13  height: 20px;
14  color: #ffffff;
15  background-color: #000000;
16 }
```

Якщо скопіювати даний кусок CSS коду в окремий файл і під'єднати його наступним чином у наш HTML документ:

Під'єднуємо CSS файл у HTML документ

```
1 <html>
2   <head>
3     <meta charset="UTF-8" />
4     <title>Приклад HTML Сторінки</title>
5     <meta name="description" value="Мета опис сторінки" />
6     <link rel="stylesheet" href="main.css" />
7   </head>
8   <body>
9     <h1>Заголовок HTML сторінки</h1>
10    <p id="paragraph">Приклад параграфа тексту</p>
11    <br />
12    <a class="my-link" href="http://google.com">Посилання</a>
```

13 `</body>
14 `</html>

(зауважте на 6-ій стрічці як ми під'єднали наші стилі із файла під назвою *main.css* з допомогою тегу *link*) і знову відкрити наш HTML файл у веб-переглядачі, тоді отримаєте щось подібне до цього:

Заголовок HTML сторінки

Посилання

Приклад параграфа тексту

Наша HTML сторінка із стилями

Синтаксис мови CSS є доволі простий і складається із блоків правил. Кожен блок правил має:

- **селектор** - ідентифікує, якому саме елементу на сторінці застосовуються правила даного блоку; у нашому вищезгаданому прикладі ми маємо 3 селектора: 1) *h1* (усі теги *h1* на сторінці), 2) *p#paragraph* (тег параграфа з ідентифікатором *paragraph*), 3) *a.my-link* (усі посилання на сторінці, що мають клас *my-link*); назва тегу, клас та ідентифікатор є одними із найпоширенішими елементами селектора, проте їх існує величезна кількість і детальніше можна ознайомитись із списком елементів, які можна використовувати у селекторі в довіднику по [CSS⁴¹](#);
- **властивості** - блок правил, які йдуть одразу після селектора і є огорнуті у фігурні дужки; вони описують які саме властивості змінити (напр. положення елементів на сторінці, кольори, розміри, шрифти, декорації, фон та межі, і т.д.) в елементі на сторінці обраного селектором, що передує даному блоку властивостей; властивостей існує величезна кількість і їх усіх можна переглянути знову ж таки в довіднику по [CSS⁴²](#); а протягом книги ми будемо розбирати саме ті із них, які пригодяться нам у нашому проекті.

Крім того селектори працюють таким чином, що можуть перебивати один одного дозволяючи писати специфічніші стилі для одних елементів і зали-

⁴¹<http://css.manual.ru>

⁴²<http://css.manual.ru>

шаючи інші елементи незмінними. Це дуже потужна властивість мови CSS, яка дозволяє з легкістю унаслідувати стилі та уникати дубляжу в коді.

Детальніше про конкретні селектори і властивості HTML елементів будемо обговорювати в процесі роботи над проектом.

Мова браузерів - Javascript

Для покращення користувачького досвіду під час користування веб-сторінками було вирішено вбудувати мову програмування у веб браузери. Після багатьох спроб і різноманітних підходів стандартом у світі веб стала мова *Javascript*⁴³. На початках використовувалась для вузького набору завдань із клієнтської валідації форм та динамізації деяких елементів. Поява технології *AJAX*, яка дозволила з допомогою мови Javascript комунікувати із сервером, цілком змінила відношення до мови і тепер ця мова переросла в одну із найпопулярніших та використовується не лише на стороні браузера, а і для програмування серверної сторони веб-аплікацій.

Ось приклад мови Javascript:

Приклад Javascript коду `lang=javascript`

```
1 window.onload = function(){
2     var mylink = document.getElementsByClassName('my-link')[0],
3         paragraph = document.getElementById('paragraph');
4     mylink.onclick = function() {
5         if (paragraph.style.visibility == '') {
6             paragraph.style.visibility = 'hidden';
7         } else {
8             paragraph.style.visibility = '';
9         }
10        return false;
11    };
12};
```

⁴³<http://uk.wikipedia.org/wiki/JavaScript>

У ньому ми, одразу після завантаження сторінки, чіпляємо подію кліку миші на наш лінк з попереднього прикладу із HTML документом із класом *my-link*. Після приєднання даного скрипта до нашої HTML сторінки параграф із текстом буде то ховатись, то показуватись.

Щоб самим потестувати даний код збережіть вище наведений кусок Javascript коду в окремий файл і під'єднайте його до нашої HTML сторінки наступним чином:

Під'єднуємо Javascript файл до HTML сторінки

```
1 <html>
2   <head>
3     <meta charset="UTF-8" />
4     <title>Приклад HTML Сторінки</title>
5     <meta name="description" value="Мета опис сторінки" />
6     <link rel="stylesheet" href="main.css" />
7     <script type="text/javascript" src="main.js"></script>
8   </head>
9   <body>
10    <h1>Заголовок HTML сторінки</h1>
11    <p id="paragraph">Приклад параграфа тексту</p>
12    <br />
13    <a class="my-link" href="http://google.com">Посилання</a>
14  </body>
15 </html>
```

На 7-ій стрічці HTML документу бачимо тег *script*, який посилається на файл *main.js*. Браузер, знайшовши даний тег, спробує піти на сервер за скриптом *main.js*, отримати його та запустити код, що у ньому знаходитьсья.

Спробуйте тепер, після підключення нашого Javascript коду, відкрити HTML документ і поклікати по посиланню. Якщо зробили все правильно, параграф тексту повинен ховатись та показуватись почергово при кожному кліку.

Незважаючи на схожість назв, мови Java та JavaScript є двома різними мовами, що мають відмінну семантику, хоча й мають схожі риси в стандартних бібліотеках та правилах іменування. Синтаксис обох мов отриманий “в спадок” від

мови С, але семантика та дизайн JavaScript є результатом впливу мов Self та Scheme.

Детальніше про мову Javascript можете почитати на [сторінці вікіпедії](#)⁴⁴. Протягом книги ми з вами будемо працювати зазвичай із додатковими бібліотеками, що значно полегшать нашу роботу на стороні браузера, а не використовувати *голий* Javascript.

Специфіка фронтенду

Усі вище перечислені технології (HTML, CSS, Javascript) складають основу так званої фронтенд (front-end, ще називають її клієнтською) веб-розробки. Іншими словами тієї частини веб-аплікації, яка безпосередньо взаємодіє із користувачем у веб-браузері.

Кожна із цих мов має затверджений стандарт і є, відповідно, реалізованою на стороні веб-переглядача. Але вже так історично склалось, що веб-браузерів на даний момент маємо далеко не один, і що ще більше погіршує ситуацію - кожен із них по різному і в різній мірі підтримує ці стандарти.

На даний момент найпопулярнішими веб-переглядачами є:

- [Chrome](#)⁴⁵,
- [Firefox](#)⁴⁶,
- [Internet Explorer](#)⁴⁷,
- [Safari](#)⁴⁸ десктопний та мобільний,
- [Opera](#)⁴⁹,
- мобільний [Android Browser](#)⁵⁰.

⁴⁴<http://uk.wikipedia.org/wiki/JavaScript>

⁴⁵<http://www.google.com.ua/intl/uk/chrome/>

⁴⁶<https://www.mozilla.org/uk/firefox/new/>

⁴⁷<http://windows.microsoft.com/uk-ua/internet-explorer/download-ie>

⁴⁸<https://www.apple.com/safari/>

⁴⁹<http://www.opera.com/uk>

⁵⁰<http://www.android.com/about/>

Таким чином те, що ви запрограмуєте на стороні клієнта і заставите чудово працювати у веб-браузері Chrome, може не зовсім коректно виглядати чи працювати у веб переглядачі Internet Explorer. І чим багатша на функціонал клієнтська сторона вашого веб-сайту, тим більша ймовірність, що вам прийдеся більше часу затратити на підпасовку вашої програми під решту веб-переглядачів.

Зазвичай власник проекту сам визначає, які переглядачі та користувачі його цікавлять. Відповідно до цих вимог вам, як розробнику, прийдеся адаптувати вашу веб-аплікацію для різних веб-переглядачів та платформ. Так, під різні платформи ваш сайт може виглядати по різному, навіть на тих самих версіях браузерів.

Лише з практикою та досвідом приходить вміння швидко вирішувати проблеми із різними версіями операційних систем та веб-переглядачів.

В той же час вже давно почали придумувати і реалізовувати масу різноманітних бібліотек та фреймворків, таких собі “надбудов” над основними мовами, які абстрагують вас від більшості відмінностей між платформами, операційними системами та різними веб-браузерами. І на даний час уже існує велика кількість бібліотек з допомогою, яких ваш HTML, CSS та Javascript код буде працювати в більшості випадків з першого разу для усіх браузерів.

У проекті даної книги ми використовуватимемо два таких додаткових інструменти:

- jQuery: Javascript бібліотека, яка дозволяє швидко реалізовувати багату на функціонал сторінку;
- Twitter Bootstrap: HTML/CSS/Javascript фреймворк, який дасть нам доступ до наперед визначених популярних на веб-сторінках віджетів; як от навігація, закладки, форми, кнопки, випадайки, лейаут сторінки і т.д.

jQuery - покращений Javascript

Якщо ми хочемо, щоб наш Javascript код працював одразу у більшості популярних веб-переглядачах, тоді нам прийдеться писати багато умовних операторів у нашому коді. Адже існує багато відмінностей в реалізації цієї мови у різних веб-браузерах, інші назви функцій, інша робота з подіями та об'єктами на веб-сторінці (тегами та атрибутами).

Щоб уникнути перевитрат часу і не винаходити ровер заново, ми будемо користуватись однією із найпопулярніших Javascript бібліотек - [jQuery⁵¹](#).

Вона не лише дозволить нам з легкістю забути про різні версії браузерів, але й даст величезну кількість додаткових функцій, що збережуть нам масу часу.

Ось лише порівняйте, як виглядатиме наш Javascript код із попереднього прикладу (де ми заставляли параграф тексту зникати на сторінці при кліку на посилання):

Javascript код з використанням jQuery

```

1 $(function(){
2     $('.my-link').click(function(){
3         var paragraph = $('#paragraph');
4         if (paragraph.is(':hidden')) {
5             paragraph.show();
6         } else {
7             paragraph.hide();
8         }
9         return false;
10    });
11 });

```

Так, рядочків коду зменшилось лише на один, проте їхня довжина значно коротша і, знаючи англійську, можна легко зрозуміти, що відбувається у кожному з них. В реальних проектах роль подібних бібліотек просто неоціненна!

⁵¹<http://jquery.com>

Twitter Bootstrap

В нашому практичному проекті ми будемо створювати багато форм, елементів навігації, кнопок, полів і тому подібних елементів. В більшості випадків такі елементи є схожі між собою на різних веб-сайтах. Саме тому на даний момент існує багато фреймворків, які дозволяють швидко будувати найчастіше використовувані елементи на веб-сторінках без проведення великої кількості часу на їхній вигляд та функціонал.

Дана книга зосереджена на веб-фреймворку Django та серверній розробці в першу чергу. В той час як ми з вами оглянемо повний цикл веб-розробки протягом практичного проекту, фронтенд розробка (зокрема верстка) не є основним акцентом даної книги. І саме тому ми скористаємося супер-популярним фреймворком [Twitter Bootstrap⁵²](#) для створення більшості графічного інтерфейсу, який працюватиме в усіх популярних веб-переглядачах.

Таким чином використовуючи певні правила при написанні HTML тегів та атрибутів ми одразу матимемо готові веб елементи на наших сторінках. Це зекономить наш час при верстці сторінок.

Підключивши Twitter Bootstrap скрипти і стилі наступний доволі простий приклад HTML сторінки:

HTML сторінка з використанням Twitter Bootstrap

```

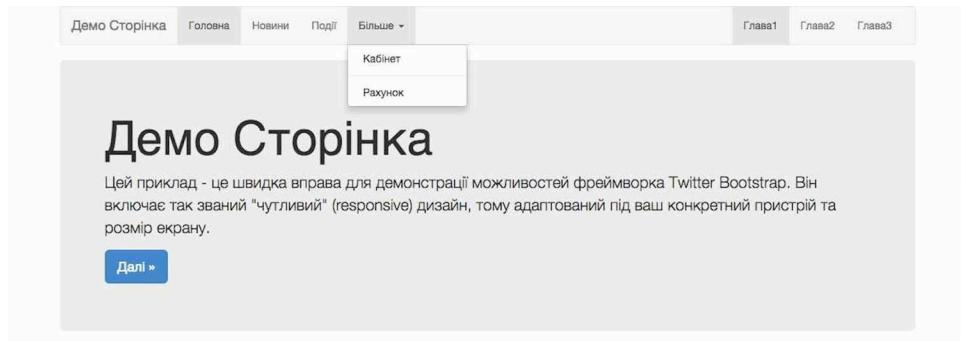
1 <html>
2   <head>
3     <meta charset="UTF-8" />
4     <title>Приклад HTML Twitter Bootstrap Сторінки</title>
5     <meta name="description" value="Мета опис сторінки" />
6     <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
7       <script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js"></script>
8       <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/js/bootstrap.min.js"></script>
9     </head>
```

⁵²<http://getbootstrap.com>

```
13 <body>
14
15     <div class="container">
16
17         <!-- Навігаційна панель -->
18         <div class="navbar navbar-default" role="navigation">
19             <div class="container-fluid">
20                 <div class="navbar-header">
21
22                     <!-- Цей елемент стане у пригоді для малих екранів -->
23                     <button type="button" class="navbar-toggle collapsed"
24                         data-toggle="collapse" data-target=".navbar-coll\
25 apse">
26                         <span class="sr-only">Toggle navigation</span>
27                         <span class="icon-bar"></span>
28                         <span class="icon-bar"></span>
29                         <span class="icon-bar"></span>
30                     </button>
31
32                     <!-- Лого -->
33                     <a class="navbar-brand" href="#">Демо Сторінка</a>
34
35             <div class="navbar-collapse collapse">
36
37                 <!-- Головна навігація -->
38                 <ul class="nav navbar-nav">
39                     <li class="active"><a href="#">Головна</a></li>
40                     <li><a href="#">Новини</a></li>
41                     <li><a href="#">Події</a></li>
42                     <li class="dropdown">
43                         <a href="#" class="dropdown-toggle"
44                             data-toggle="dropdown">Більше
45                             <span class="caret"></span></a>
46                         <ul class="dropdown-menu" role="menu">
47                             <li><a href="#">Кабінет</a></li>
```

```
48          <li class="divider"></li>
49          <li><a href="#">Рахунок</a></li>
50      </ul>
51  </li>
52</ul>
53
54      <!-- Користувачькі лінки -->
55      <ul class="nav navbar-nav navbar-right">
56          <li class="active"><a href="#">Глава1</a></li>
57          <li><a href="#">Глава2</a></li>
58          <li><a href="#">Глава3</a></li>
59      </ul>
60
61      </div>
62  </div>
63</div>
64
65      <!-- Main component for a primary marketing message or call to\
66 action -->
67      <div class="jumbotron">
68          <h1>Демо Сторінка</h1>
69          <p>Цей приклад - це швидка вправа для демонстрації можливост\
70 ей фреймворка Twitter Bootstrap. Він включає так званий "чутливий" (\\
71 responsive) дизайн, тому адаптований під ваш конкретний пристрій та \
72 розмір екрану.</p>
73          <a class="btn btn-lg btn-primary" href="#" role="button">
74              Далі &raquo;</a>
75          </p>
76      </div>
77
78  </div>
79
80</body>
81</html>
```

набуде ось якого вигляду у вашому веб-браузері:



Приклад Twitter Bootstrap сторінки

Також спробуйте зменшити ширину вікна вашого веб-переглядача і побачите як класно верстка сторінки адаптується під нього.

Таким чином із допомогою Twitter Bootstrap ми з легкістю отримали:

- навігаційну панель із лого, основною навігацією та випадайкою, а також другорядною навігацією;
- тіло сторінки із гарно застиленим текстом та кнопкою “Далі”.

і при цьому лише написали HTML код із потрібними тегами та класами. Крім того ця сторінка гарно виглядає і на вузьких екранах. Лише уявіть скільки часу нам потрібно було б, щоб зверстати її з нуля, додати динаміки, заставити виглядати добре у різних браузерах та вужчих екранах.

Саме тому ми використовуватимемо даний фреймворк.

На [сайті бібліотеки⁵³](http://getbootstrap.com) можна детальніше ознайомитись із усіма доступними віджетами та функціоналом.

Бекенд

До цього моменту ми розглянули ту кухню, яка готове нам картинку у веб-переглядачі: HTML, CSS, Javascript та деякі надбудови над ними.

⁵³<http://getbootstrap.com>

Але, звісно, часи, коли HTML код писали вручну в статичних файлах і клали їх на сервері у папочки давно минули. В даний час більшість сучасних веб-сайтів в інтернеті генерується на льоту з допомогою мов програмування, веб-фреймворків та систем менеджменту вмісту (CMS - Content Management System).

Саме тому наші з вами знання, як веб-девелоперів, не обмежуються мовами, що використовуються вбраузері при верстці HTML сторінки. Однаково важливими є усі ті технології, що дають нам можливість мати динамічні веб сайти із багатим функціоналом, отримувати та запам'ятувати дані від користувача:

- *база даних*: уже із назви зрозуміло, що цей інструмент дозволяє зберігати та отримувати дані для веб-аплікації; використовувати ми будемо реляційну базу даних MySQL;
- *серверна мова програмування*: мова, якою ми отримуватимемо і оброблятимемо запити від користувачів, працюватимемо із базою даних та генеруватимемо HTML код для наших веб-сторінок; у нашому випадку це, звісно, буде мова програмування Python.

Бекенд (з англ. “back-end”) ще називають серверною стороною. Тобто та частина веб-сайту, яка відповідає за обслуговування запитів користувача та генерацію коду для веб-сторінки.

Мова програмування Python

Більше, ніж половина нашого з вами часу буде присвячена написанню коду саме на мові Python.

З допомогою Python ми:

- організуємо структуру веб-адрес аплікації;
- налаштуємо проект;
- реалізуємо збереження та отримання даних із бази;
- запрограмуємо усі функціональні сторінки нашого веб-сайту;
- і ще багато іншого.

Ця мова є кросплатформеною та широкого призначення (звичайно ми її використовуватимемо супер для веб-розробки). Мова Python є динамічною інтерпритованою мовою з простим синтаксисом, що дає можливість програмісту надзвичайно швидко створювати програми та фокусуватись на вирішенні кінцевих проблем.

Ось приклад Python коду з реального Django проекту, який підтверджує, що дана мова є дійсно простою, лаконічною та зрозумілою:

Приклад Python коду із Django проекту

```
1 #stat_payments
2 class StatsPayments(TemplateView):
3     template_name = 'aikido_school/stats_payments.html'
4
5     @method_decorator(permission_required('aikido_school.stats_payments'))
6
7     def dispatch(self, *args, **kwargs):
8         return super(StatsPayments, self).dispatch(*args, **kwargs)
9
10    def get_context_data(self, **kwargs):
11        context = super(StatsPayments, self).get_context_data(**kwargs)
12
13
14        year = None
15        month = None
16        filter = None
17        object_list = Group.objects.all()
18        if self.request.GET.get('month') and self.request.GET.get('year'):
19
20            year = self.request.GET.get('year')
21            month = self.request.GET.get('month')
22            filter = Q(student__fee__fee_date__year=year) &
23                      Q(student__fee__fee_date__month=month)
24
25            if filter is not None:
26                object_list = object_list.filter(filter)
27
28            object_list = object_list.annotate(sum=Sum('student__fee__pr')
```

```

27     ice'),
28             count=Count('student__fee__price'))
29
30         form = StatsPaymentsForm(self.request.GET or None, initial={
31             'month': datetime.today().month, 'year': datetime.today(\n
32         ).year})
33
34         context['object_list'] = object_list
35         context['one_column'] = True
36         context['form'] = form
37         if year and month:
38             context['year'] = year
39             context['month'] = _(month_name[int(month)])
40
41     return context

```

Якщо ви взялися за дану книгу, то це означає, що у вас уже є мінімальна база з мови програмування Python. Якщо ж ні, тоді зверніться до [Вступу](#) та перегляньте наданий там список матеріалів з необхідним мінімумом інформації.

Як тест для ваших знань спробуйте переглянути та зrozуміти поверхнево (адже код працює у фреймворку Django, який нам ще треба буде освоїти) логіку та конструкції мови Python у класі наведеному вище.

Синтаксис мови, ООП та інші базові концепції мови ми не будемо розбирати в процесі проекту, а звертатимемо увагу лише на найважливіші аспекти веб-програмування, Django і комунікацію між сервером та веб-переглядачем.

База даних MySQL

Для зберігання даних нашої аплікації ми спочатку скористаємося простою базою даних *sqlite3*, а вже пізніше переключимось на реляційну базу даних [MySQL](#)⁵⁴. Це одна із найпоширеніших баз даних, яку ми можемо безкоштовно використовувати у наших цілях.

⁵⁴<http://uk.wikipedia.org/wiki/MySQL>

Ця база даних дасть нам можливість:

- зберігати дані;
- робити пошук серед даних;
- та, звісно, отримувати дані у потрібному форматі.

Веб-фреймворк Django дасть нам усі необхідні інструменти для роботи з даними таким чином, що нам практично не доведеться мати справу напряму із SQL запитами. Більшість завдань із даними ми зможемо реалізувати не відходячи від мови Python.

Тому в книзі ми лише поверхнево оглянемо мову запитів реліційної бази даних - SQL. Короткий огляд найбільш необхідних інструментів та команд при роботі з базою MySQL можете додатково знайти у конспекті-шпаргалці, що йде разом із *Рекомендованым* пакетом книги.

Веб-фреймворк Django

Згодом, коли використання мови програмування для генерації HTML сторінок уже стало буденною річчю, програмісти не зупинились і продовжили удосконалювати процес створення веб-сайтів. Після кожного завершеного веб-проекту набуті навички, підходи та код переносились та адаптувались під нові проекти таким чином перетворюючись на маленькі фреймворки для швидкого створення нових веб-аплікацій. Адже для чого придумувати колесо, якщо можна використати попередній код перед цим зробивши його універсальнішим.

Веб-фреймворк - це набір інструментів, бібліотек, аплікацій, заготовок, що забезпечують стандартний та найбільш затребуваний набір функціоналу для переважної більшості веб-аплікацій та сайтів.

Одним із найпоширеніших веб-фреймворків на мові Python є [Django⁵⁵](https://www.djangoproject.com/). Він дозволяє із блискавичною швидкістю створювати прототипи веб-сайтів і надає необхідний мінімум функціоналу:

⁵⁵<https://www.djangoproject.com/>

- *моделі*: місток між Python класами та базою даних, який робить усю складну роботу щодо збереження та пошуку даних для вас;
- *шаблони*: генерація HTML коду без використання Python мови, натомість із мінімальними вкрапленнями логіки з допомогою динамічних Django тегів напряму у HTML код; це дає можливість нам мати так званий **MVC⁵⁶** підхід та розділяти логіку від даних та представлення;
- *диспетчер URL*: інструмент, з допомогою якого можемо легко будувати ієрархію URL адрес нашого веб-сайту;
- *адміністративна частина*: іде разом із Django та дозволяє керувати даними веб сайту та його налаштуваннями без додаткової розробки;

The screenshot shows the Django Admin dashboard. At the top, there's a header bar with the text "Django адміністрування" and "Вітаємо, admin. Змінити пароль / Вийти". Below the header, there's a sidebar titled "Адміністрування сайта" which lists various models with "Добавити" (Add) and "Змінити" (Change) buttons. The sidebar includes sections for "Додатки Alkido_School" (Атестації, Внески, Групи, Місочні Журнали, Результати, Семінари, Студенти), "Додатки Auth" (Users, Групи), and "Додатки Sites" (Сайти). To the right of the sidebar, there's a "Недавні дії" (Recent Actions) box containing "Мої дії" and "Немас". The main content area is currently empty.

Django адміністративна частина

- *користувачі*: якщо ваша веб-аплікація потребує логованих користувачів, тоді нічого додаткового практично не доведеться розробляти; Django дає весь функціонал, що дозволить вам мати дані користувачів, форми логування, реєстрації та усі налаштування з безпеки, ролі та дозволи для користувачів;
- *форми*: додаткові Python класи для роботи із веб-формами; вони економлять масу часу при розробці стандартних форм;
- *розробницькі інтерфейси (Development APIs)*: найпоширенішим є **REST⁵⁷**; з Django можна легко навчити вашу веб-аплікацію “говорити” у форматі REST, який часто буває корисним для мобільних додатків та комунікації із іншими сервісами;

⁵⁶<http://bit.ly/vpmvc>

⁵⁷<http://uk.wikipedia.org/wiki/REST>

- *міграція*: такий інструмент як South дозволяє легко змінювати та мігрувати ваші дані в базі, їхній формат та вигляд, а також робити це повторно без будь-яких проблем;
- як і у будь-якому іншому веб-фреймворку такі речі як деплоймент на кінцевий сервер, документація, автоматичні тести, атомарна розробка (модульність), розробницькі інструменти, є добре продумані у фреймворку Django; вони допоможуть вам швидше та з легкістю закінчувати ваші веб-проекти.

Крім того, що можна користуватись усіма вбудованими функціями фреймворку, також існує маса бібліотек та аплікацій.

Ну і ще один не менш важливий аргумент за Django - це OpenSource фреймворк. Його код можна вільно використовувати, читати та змінювати.

В більшості випадків напряму змінювати код OpenSource бібліотек у власних проектах є поганою ідеєю. Це в майбутньому дасть вам багато проблем при спробі оновлення проекту до нової версії цієї бібліотеки. Натомість Python є настільки динамічною мовою, що дозволяє практично будь-які зміни у код робити ззовні, із власного коду. Так званий “monkey patching”.

Репозиторій коду Git

Ця секція немає напряму відношення до веб-розробки. Репозиторій коду - це швидше потреба для кожного проекту з програмним кодом. Особливо, якщо над ним працює більше, ніж одна людина.

Репозиторій коду - це місце збереження вихідного коду та інших матеріалів програмного продукту, яке забезпечує правильний формат зберігання та:

- одночасну роботу двох і більше людей над проектом;
- історію змін в коді;

- одночасну роботу над кількома різними релізами та функціоналом незалежно один від одного;
- резервну копію коду.

Наш проект ми будемо зберігати в репозиторії коду [Git⁵⁸](#). Протягом останніх років він набув неабиякої популярності як мінімум в OpenSource спільнотах та проектах. А публічний сервіс для зберігання проектів в репозиторіях Git - [github.com⁵⁹](https://github.com) став передовим для OpenSource проектів.

Під час роботи над проектом ми регулярно зберігатимемо зміни в репозиторії притримуючись кращих практик. Тому по завершенню книги ви не лише володітимете азами Django фреймворку та веб-розробки, але й непогано працюватимете із репозиторієм.

Для загального ознайомлення із принципом роботи репозиторію коду рекомендую ознайомитись із статею про Git на мому блозі: [Що таке репозиторій коду або ЛікБез по Git⁶⁰](#).

Також із Рекомендованим пакетом книги йде конспект-шпаргалка щодо використання репозиторію Git.

Домашнє завдання

Отже, в даній главі ми коротко пройшлися по основних аспектах веб-розробки. Оглянули принцип комунікації сервер-клієнт, протокол HTTP, фронтенд розробку (HTML, CSS, Javascript, jQuery, Twitter Bootstrap), серверну сторону (Python, MySQL, Django), репозиторій коду Git.

Протягом розробки нашого з вами практичного проекту ми будемо детальніше розбирати потрібні нам аспекти кожної із цих технологій. Але, якщо ви поки лише початківець у веб-розробці, тоді перед переходом до наступної глави рекомендую детальніше ознайомитись із ними. Це дасть вам можливість краще засвоїти матеріал книги.

Отже, на домашнє завдання затратьте кілька годин і:

⁵⁸<http://uk.wikipedia.org/wiki/Git>

⁵⁹<https://github.com/>

⁶⁰<http://www.vitaliyopodoba.com/2014/06/git-basics/>

- знайдіть та пройдіть невеличкий онлайн туторіал або курс по мові HTML на зручній для вас мові; в інтернеті є маса доступних матеріалів як на англійській так і на російській та українській мовах;
- те ж саме із CSS; достатньо навіть вище згаданих посилань на статті у вікіпедії;
- ознайомтеся із мовою Javascript;
- також із мовою запитів реляційних баз даних: SQL;
- перечитайте та поекспериментуйте із Лікбезом з Git репозиторією;
- і загалом, усі наведені посилання на зовнішні ресурси у даній главі є важливими для подальшого ознайомлення.

Python не згадав, адже це само собою розуміється. Без нього нікуди.

Усю теорію, що набудете в результаті виконання даного домашнього завдання ми з вами успішно застосуємо на практиці в подальших главах. Також не забувайте одразу експериментувати із прикладами при проходженні та засвоєнні теоретичних матеріалів.

...

У наступній главі ми дамо визначення нашому практичному проекту, сформулюємо основні завдання та специфікації, а також перелічимо усі ті навички, що отримаємо в результаті імплементації даного проекту.

3. Проект: база даних для обліку студентів

Коли я починав роботу над даною книгою, то перша ідея була взяти за проект розробку персонального блога програміста. І вже написавши код для близько половини проекту я зрозумів, що він не є показовим та бракує специфікацій для важливих аспектів веб-розробки, як от робота з формами, автентифіковані користувачі, інтенсивна робота з базою даних і ще кілька речей, які, я вважав, є необхідними навіть для початківців.

На той час я вже мав кількох студентів на персональному менторстві за напрямком веб-розробка з Python та Django, а також мав невеликий список типових задач, що дають Python джуунікам.

Тому, щоб не видумувати велосипед, я взяв найпоширеніший тестовий проект, розширив його деяким додатковим функціоналом таким чином, щоб в кінцевому результаті надати необхідний мінімум знань та практики початківцю.

Список тестових проектів у мене був ще від часів, коли проходив інтерв'ю на інших фірмах, а також деято із студентів уже пробували подаватись на позиції джууніорів і ділились зі мною своїм досвідом співбесіди. Ці ж задачки я час до час використовую під час пошуку джууніора Python програміста у свою власну фірму. Зазвичай такий тестовий проект я даю кандидату на один-два тижні розробки, взалежності від складності проекту.

Специфікації проекту

У цій книзі ми зайдемось розробкою бази даних для обліку студентів. Основними сутностями цієї бази будуть студенти та групи.

Кожен студент матиме мінімальний набір необхідних полів (усі поля є обов'язковими):

- повне ім'я: ім'я, прізвище та по-батькові;
- дата народження;
- фото студента;
- номер студентського білету;
- група;
- та додаткові нотатки.

Група в свою чергу матиме:

- називу (обов'язкове поле);
- старосту групи - одного із студентів (не обов'язкове поле);
- та додаткові нотатки.

Наша аплікація дозволятиме:

- додавати нових студентів та групи;
- редагувати існуючих студентів та групи;

Форма редагування Студента

The screenshot shows a web-based form titled "Форма редагування Студента". The form fields include:

- Ім'я*: Віталій
- По-батькові: Іванович
- Прізвище*: Подоба
- № Білету*: 254
- Дата народження: 1984-06-17
- Фото: Наразі: student_photos/None_podoba3_1.jpg [Remove] [Clear]
- Змінити: Choose File [No file chosen]
- Група*: MтM - 21
- Додаткові Нотатки: тест d

At the bottom of the form are two buttons: "Зберегти" (Save) and "Скасувати" (Cancel).

Форма редагування студента

- видаляти існуючих студентів та групи;
- переглядати списки існуючих студентів та груп, а також переглядати студентів в контексті обраної групи;

Сервіс Обліку Студентів

Група: Усі Студенти

Студенти Відвідування Групи

Студент успішно доданий.

База Студентів

#	Фото	Прізвище ↑	Ім'я	№ Білету	Дії
1		Корост	Андрій	2123	<button>Дія ▾</button>
2		Подоба	Віталій	254	<button>Дія ▾</button>
3		Припутла	Тарас	5332	<button>Дія ▾</button>

© 2014 Сервіс Обліку Студентів

Сторінка із списком студентів

- вести журнал присутності студентів;

Сервіс Обліку Студентів

Група: Усі Студенти

Студенти Відвідування Групи

Облік Відвідування

← Жовтень 2014 →

#	Студент	Ср 1	Чт 2	Пт 3	Сб 4	Нд 5	Пн 6	Вт 7	Ср 8	Чт 9	Пт 10	Сб 11	Нд 12	Пн 13	Вт 14	Ср 15	Чт 16	Пт 17	Сб 18	Нд 19	Пн 20	Вт 21	Ср 22	Чт 23	Пт 24	Сб 25	Нд 26	Пн 27	Вт 28	Ср 29	Чт 30	Пт 31
1	Подоба Віталій	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>						
2	Корост Андрій	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		
3	Припутла Тарас	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>																					

© 2014 Сервіс Обліку Студентів

Журнал відвідуваності студентів

- працювати лише залогованим користувачам;
- запускати окремий скрипт в командній стрічці, щоб отримати список студентів у вказаній групі;

Крім того вимоги проекту включатимуть:

- адаптацію усього інтерфейсу нашої веб-аплікації під українську мову засобами інтернаціоналізації в Django;
- виділення частину коду проекту в окрему аплікацію з подальшою можливістю використання коду в інших проектах;
- автоматичний тест для одного користувачького сценарію;
- ведення журналу щодо створення, редагування та видалення студентів і груп.

Ну і зрозуміло, що наприкінці виконаної роботи потрібно буде перенести проект і налаштувати програму на кінцевому сервері.

Чого ви навчитеся протягом проекту?

Таким чином після реалізації проект згідно вище зазначених специфікацій ми з вами навчимось:

- налаштовувати своє робоче середовище, створювати проект та аплікації в Django;
- працювати з базою даних з допомогою Django моделей;
- верстати форми, елементи навігації та динамізувати сторінки з допомогою HTML, CSS, Twitter Bootstrap, Javascript бібліотеки jQuery та технології Ajax⁶¹;
- будувати Django в'юшки (від англ. view - вид, бачити) та працювати із динамічними шаблонами;
- перекладати (інтернаціоналізувати) інтерфейс веб-аплікації;
- будувати структуру веб-адрес сайту в Django з допомогою диспетчера адрес;
- писати власну Django “мідлавару” (middleware⁶²);
- створювати Django процесор контексту;

⁶¹<http://uk.wikipedia.org/wiki/AJAX>

⁶²<http://bit.ly/vpmiddleware>

- писати Django команду;
- реалізувати автоматичний тест;
- працювати із Django сигналами (подіями) та Python модулем логування;
- створювати *кастомний* шаблонний тег;

Слово “*кастомний*” (від англ. custom) я буду часто вживати у даній книзі разом із словами “код”, “рішення”, “проект”. Воно може вживатись у одному із наступних значень: власний, окремий, під замовлення або “на ключ”. Тобто *кастомний код* - це код, який спеціально написаний під даний проект. Також, “*кастомізувати*” означатиме змінювати дефолтну поведінку чи функціонал, зазвичай в Django.

- працювати із Django адмінкою та робити невеликі зміни до її інтерфейсу;
- обробляти веб-форми як з допомогою кастомного Python коду на сервері так і з допомогою заготовлених в Django класів форм;
- використовувати вбудовану в Django систему авторизації та автентифікації, щоб дозволити лише залогованим користувачам використовувати нашу аплікацію;
- деплоїти проект на продакшин сервер.

Продакшин сервер (з англ. production - виробництво) - це кінцевий комп’ютер, на якому встановлений програмний продукт для його використання кінцевими користувачами.

Деплоймент (з англ. deployment - розгортання) - це процес, що робить програмний продукт доступним для його користувачів. Зазвичай він полягає у перенесенні коду програми на кінцевий сервер (так званий “продакшн”), звідки він обслуговує кінцевих користувачів.

Що не входить у даний проект?

Завдяки роботі над проектом у даній книзі ви набудете усіх необхідних знань, щоб претендувати на посаду джуніора веб-розробника на Django фреймворку.

Даний фреймворк покриває і більш обширні та складніші теми, які є важливими для освоєння при роботі над великими та високо-навантаженими проектами. Але початківцю вони зазвичай не потрібні та і є занадто складними для перших кроків. Відповідно їх зазвичай не вимагають від джуніорів.

Тому дані теми не входять у дану книгу та швидше за все будуть розкриті у наступних моїх навчальних матеріалах, уже для тих, хто осягнув основи веб-розробки з допомогою Django:

- monkey patching⁶³ в контексті Django;
- реалізація кастомних полів для Django моделей;
- поглиблена кастомізація адмін частини Django;
- оптимізація швидкодії Django сайту;
- робота на низькому рівні із базою даних (SQL запити, транзакції, оптимізація швидкодії при роботі з базою);
- безпека при розробці на Django;
- кешування в Django;
- реалізація REST API;
- автоматичне тестування, юніт тести і т.д. лише поверхнево оглянемо на прикладі тесту одного користувацького сценарію поведінки.

⁶³https://ru.wikipedia.org/wiki/Monkey_patch

Частина даних тем є розкритою в серії постів у мене на блозі: *Кращі практики розробки з Django⁶⁴*. В майбутньому я планую присвятити окремі серії постів таким темам як безпека та швидкодія в Django.

Домашнє завдання

Тепер ви уже можете собі уявити, що ми з вами будуватимемо та який об'єм робіт на нас чекає. На перший погляд списки вимог досить довгі, особливо як для початківців. Але не варто засмучуватись, адже фреймворк Django для кожного із завдань підготував для нас цілий ряд інструментів, які дуже спростятауть наше з вами життя.

Професійний програміст, який знається на розробці під Django, може зазвичай виконати даний проект в межах 2-3х днів. Джуніор в межах 1-2х тижнів. А початківець, який з нуля вивчає веб-програмування та фреймворк Django, від 1-го до бти місяців (за умови, що вже володіє азами серверної мови програмування). Принаймні так показує мій досвід роботи із початківцями.

На домашнє завдання цієї глави пропоную вам детальніше переглянути зображення нашої applікації, що наведені вище та подумати як можна їх покращити, у чому бачите недоліки.

А також відвідайте сторінку Django у вікіпедії, зокрема секцію “*Можливості⁶⁵*” та ознайомтесь детальніше із набором інструментів та функціоналу, який даний фреймворк пропонує одразу по встановленні. Якщо щось не буде зрозуміло - нічого страшного, це завдання швидше для ознайомлення, щоб ви могли зорієнтуватись в можливостях Django.

...

В наступній главі ми вже нарешті переходимо до практики. Налаштуємо наше робоче середовище, підготуємо редактор коду, заінсталюємо необхідні інструменти, створимо свій перший Django проект і побачимо дефолтну сторінку applікації. Тому пропоную зробити паузу, приділити час та увагу

⁶⁴<http://www.vitaliyopoda.com/2014/07/django-dev-env-hints>

⁶⁵<http://bit.ly/vpdjangofeatures>

домашньому завданню, запастись чаєм чи кавою, а вже тоді рухатись далі до наступної глави.

4. Робоче середовище та перший Django проект

У цій главі ми переходимо врешті решт до практики. А саме до розбору нашого з вами робочого середовища та інструментів, з допомогою яких реалізовуватимемо Django проект.

Спосіб інсталяції та конфігурації необхідних програм на вашій розробницькій машині буде часом відрізнятись від того, як ми це будемо пізніше робити на продакшин сервері. Адже задача цієї глави - це подати найшвидший спосіб підготовки вашого власного робочого середовища. Єдина вимога - це зберегти основні (мажорні) версії інструментів, які ми встановлюватимемо, щоб не було проблем при перенесенні коду на кінцевий сервер.

У секціях, що підуть далі, я намагатимусь розглядати кожен інструмент в контексті трьох операційних систем: Linux (дистрибутив Ubuntu), Mac OS та Windows.

Операційна система

Я працював із Django на трьох операційних системах: Linux, Windows, Mac OS. Проте, мушу визнати, що найменше потішила робота на ОС Windows. Розробницькі, демо та продакшин сервери, що їх ми підтримуємо для наших клієнтських проектів, на даний момент, усі використовують ОС Linux, а саме дистрибуцію [Ubuntu](#)⁶⁶. У своїй щоденній роботі під час розробки я використовую Mac OS і також є цілком задоволений, адже дана операційна система надає близькі до Linux інструменти розробки.

⁶⁶<http://uk.wikipedia.org/wiki/Ubuntu>



Мій старий добрий Макбук

Також з досвіду роботи з початківцями можу сказати, що найважче було почати тим, хто пробував працювати на Windows. Численні відмінності, різні версії системи, "взламані" інсталяції - все це спричиняло до різноманітних проблем на першому кроці щодо налаштування робочого середовища та інсталяції основних бібліотек. Часом це забирало навіть до 2х тижнів. В той час як ті, хто одразу ставив Ubuntu і інсталював увесь необхідний інструментарій там, могли впоратись в межах 3-4х днів. І це включно з базовим розбором нової для них операційної системи.

Тому, якщо в даний момент ви користуєтесь виключно ОС Windows, рекомендую спробувати поставити собі [Ubuntu](http://www.ubuntu.com/)⁶⁷ паралельно з Windows та переключатись між ними при потребі. Інсталятор Ubuntu дозволить вам обирати операційну систему для завантаження на етапі запуску вашого комп'ютера.

В більшості випадків під час розробки реальних проектів на Django ви матимете справу із ОС Linux. Особливо при деплойменті на кінцевий сервер. Це є ще однією причиною чому варто саме зараз розглянути перехід

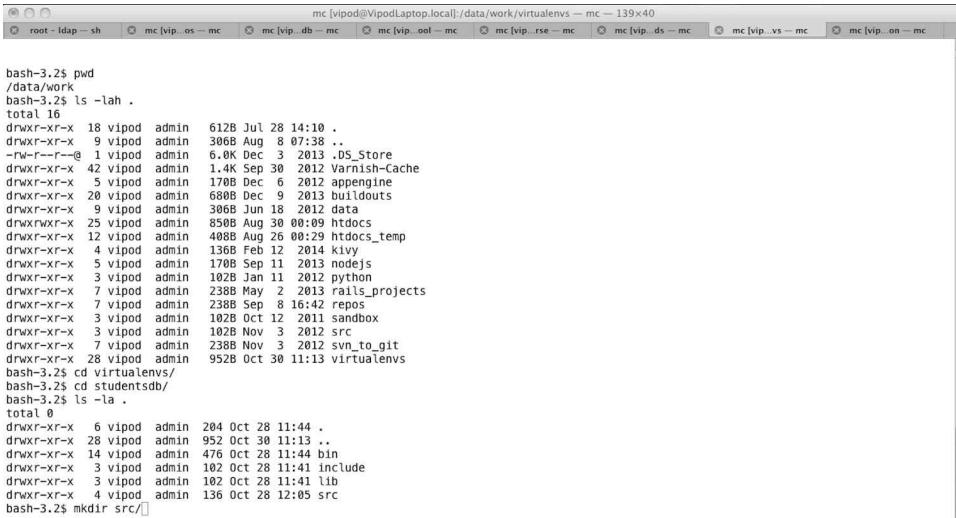
⁶⁷ <http://www.ubuntu.com/>

із Windows на Linux (оптимально Ubuntu) та по-трохи почати освоювати нову для вас операційну систему. Знання Linux буде ще одним плюсом для вас як початківця на співбесіді.

Усі приклади та код, що наведений у даній книзі написано на Mac OS та протестовано на ОС Linux (дистрибутив Ubuntu). Якщо при роботі над даним проектом використовуючи іншу операційну систему, ви отримуватимете несподівані проблеми, тоді звертайтесь у закриту групу підтримки за порадою.

Командна стрічка

Інструмент номер один без якого програмісту нікуди - *командна стрічка*. Якщо ви взялися за мову Python, тоді швидше за все прийдеться взятись за операційну систему Linux. Якщо ви взялися за операційну систему Linux, тоді швидше за все вам прийдеться на базовому рівні освоїти командну стрічку.



```

mc [vipod@VipodLaptop.local]:/data/work/virtualenvs — mc — 139x40
moot - ldap - sh   mc [vip.. os - mc  mc [vip.. db - mc  mc [vip.. oot - mc  mc [vip.. rse - mc  mc [vip.. ds - mc  mc [vip.. vs - mc  mc [vip.. on - mc

bash-3.2$ pwd
/data/work/
bash-3.2$ ls -lah .
total 16
drwxr-xr-x  18 vipod  admin  612B Jul 28 14:10 .
drwxr-xr-x  9 vipod  admin  306B Aug  8 07:38 ..
-rw-r--r--@  1 vipod  admin  6.0K Dec  3 2013 .DS_Store
drwxr-xr-x  42 vipod  admin  1.4K Sep 30 2012 Varnish-Cache
drwxr-xr-x  5 vipod  admin  170B Dec  6 2012 appengine
drwxr-xr-x  20 vipod  admin  680B Dec  9 2013 buildouts
drwxr-xr-x  9 vipod  admin  306B Jun 18 2012 data
drwxrwxr-x  25 vipod  admin  850B Aug 30 00:09htdocs
drwxr-xr-x  12 vipod  admin  408B Aug 26 00:29htdocs_temp
drwxr-xr-x  4 vipod  admin  136B Feb 12 2014 kivy
drwxr-xr-x  5 vipod  admin  170B Sep 11 2013 nodejs
drwxr-xr-x  3 vipod  admin  102B Jan 11 2012 python
drwxr-xr-x  7 vipod  admin  238B May  2 2013 rails_projects
drwxr-xr-x  7 vipod  admin  238B Sep  8 16:42 repos
drwxr-xr-x  3 vipod  admin  102B Oct 12 2011 sandbox
drwxr-xr-x  3 vipod  admin  102B Nov  3 2012 src
drwxr-xr-x  7 vipod  admin  238B Nov  3 2012 svn_to_git
drwxr-xr-x  28 vipod  admin  952B Oct 30 11:13 virtualenvs
bash-3.2$ cd virtualenvs/
bash-3.2$ cd studentsdb/
bash-3.2$ ls -la .
total 0
drwxr-xr-x  6 vipod  admin  204 Oct 28 11:44 .
drwxr-xr-x  28 vipod  admin  952 Oct 30 11:13 ..
drwxr-xr-x  14 vipod  admin  476 Oct 28 11:44 bin
drwxr-xr-x  3 vipod  admin  102 Oct 28 11:41 include
drwxr-xr-x  3 vipod  admin  102 Oct 28 11:41 lib
drwxr-xr-x  4 vipod  admin  136 Oct 28 12:05 src
bash-3.2$ mkdir src/_

```

Terminal на Macintosh

Якщо навіть вам вдасться обійтись без командної стрічки на вашому локальному комп’ютері використовуючи круті сучасні інструменти для розробки, то швидше за все, коли настане момент релізу вашого коду на кінцевий сервер, ви зрозумієте, що вам неабияк бракує вмінь управління комп’ютером використовуючи лише командну стрічку. Адже більшість серверів не мають десктопної версії (графічний інтерфейс користувача) заїнсталеної та вимагають роботи або через спеціальну контроль панель (яка далеко не у всіх випадках задовільняє вашим потребам) або через командну стрічку, консоль.

Тому рекомендую одразу почати робити базові речі в командній стрічці. Це буде також плюсом у вашому резюме, адже це означає, що ви “продвинутий” Linux користувач.

Основні речі, що ми будемо виконувати у командній стрічці:

- навігація папками;
- створення, редагування та видалення файлів;
- запуск скриптів та команд.

Linux та Macintosh

На Лінуксі та Макінтоші у вас одразу вже все є готове для використання. Це програма Terminal. Запускаєте і користуєтесь. Маєте закладки (Ctrl-T), історія (Ctrl-R), підсвідка (в деяких випадках потрібно налаштувати або навіть доставити додаткові пакети).

Опціонально можете заінсталювати редактор для режиму командної стрічки (тобто без графічного інтерфейсу), який допоможе краще працювати із файлами. Це може бути:

- nano;
- joe;
- vim або vi (він зазвичай стоїть одразу на обидвох операційних системах).

Два перші редактори потрібно заінсталювати. На Лінуксі це робиться командою:

Інсталяція редактора nano на Ubuntu

```
1 $ sudo apt-get install nano
```

В прикладах із командними стрічками символ долара ("\$") не потрібно копіювати. Це запрошення вашої командної стрічки до вводу тексту.

Увага! На Linux та Mac OS не рекомендується працювати під адміністративним користувачем root. Натомість лише в командах, які вимагають адміністративних доступів (напр. інсталяція глобальних програм та бібліотек чи створення користувачів), потрібно користуватись командою sudo, щоб виконувати ту чи іншу команду від імені root користувача. Користуйтесь sudo лише у тих випадках, де приклад в книзі також використовує sudo. В усіх інших випадках не користуйтесь sudo, хіба що ви дійсно знаєте що робите. Напр. якщо створити папку від імені root, тоді в ній неможливо

буде коректно працювати під вашим стандартним користувачем. Завжди слідкуйте під яким користувачем ви працюєте!

На макінтоші все трохи складніше, треба спочатку поставити порти, які вже інсталюватимуть додаткові бібліотеки подібним чином.

Якщо ви використовуєте Mac OS, тоді першим ділом поставте собі так звані порти, з допомогою, яких ви далі зможете встановлювати інші бібліотеки та інструменти. Я особисто використовував спочатку Mac Ports⁶⁸, але згодом перейшов на Homebrew⁶⁹. Обидва ці менеджери пакетів для Макінтоша значно спростять вам життя.

Інсталяція редактора nano на Mac OS

```
1 # використовуємо Mac Ports менеджер
2 # зауважте використання команди sudo для виконання port під кореневим
3 # користувачем (адміном)
4 $ sudo port install nano
5
6 # або використовуючи homebrew
7 $ brew install nano
```

Windows

На ОС Windows є “cmd”. Але, як на мене, користування ним досить непросте. Тому рекомендую натомість зainсталювати програму [PowerShell⁷⁰](#) і отримувати більше задоволення при роботі із командною стрічкою на Віндовсі.

⁶⁸<https://www.macports.org/>

⁶⁹<http://brew.sh/>

⁷⁰<http://technet.microsoft.com/ru-RU/scriptcenter/dd742419.aspx>

Менеджер файлів

Ще коли працював на Windows, то на власній машині користувався менеджером файлів [Total Commander](#)⁷¹. Це дозволяло мені швидше робити інтенсивну навігацію по файловій системі та реорганізацію файлів. Стандартний Windows Explorer навіть із шорткатами (з англ. shortcut - скорочення, швидкі клавіші доступу до функцій програми) не дозволяв мені так швидко працювати із файловою системою.

Згодом, перейшовши на Лінукс я почав користуватись файл менеджером [Midnight Commander](#)⁷².

Пізніше при переїзді на Макінтош я залишився далі із Midnight Commander заінсталювавши його через Mac Ports (можна також це зробити через Homebrew).

Такий дво-віконний файл менеджер у поєднанні із командною стрічкою дає мені можливість швидко вирішувати усі мої потреби при роботі із папками та файлами.

У відео уроках, що йдуть із Рекомендованим пакетом даної книги, ви можете бачити як я поєдную командну стрічку та менеджер файлів під час своєї роботи.

Даний інструмент є цілком опціональним і вам вирішувати чи потрібен він вам. Якщо ж хочете спробувати, тоді дивіться нижче коротенькі інструкції з інсталяції у різних операційних системах:

Linux (Ubuntu) - Midnight Commander

Для інсталяції Midnight Commander з допомогою менеджера файлів aptitude потрібно спочатку додати новий репозиторій до вашої бази, а вже тоді запустити команду інсталяції:

⁷¹<http://www.ghisler.com/>

⁷²http://uk.wikipedia.org/wiki/Midnight_Commander

Інсталюємо Midnight Commander на Ubuntu

```
1 # додаємо посилання на новий репозиторій, де і є наш менеджер файлів
2 $ sudo add-apt-repository ppa:eugenesan/ppa
3
4 # оновляємо список доступних пакетів з нового репозиторія
5 $ sudo apt-get update
6
7 # ну і тепер сама інсталяція
8 $ sudo apt-get install mc
```

Mac OS - Midnight Commander

Цей файловий менеджер доступний серед портів у обидвох менеджерів пакетів: Homebrew і Mac Ports.

Інсталюємо Midnight Commander на Macintosh

```
1 # з допомогою Homebrew
2 $ brew install mc
3
4 # з допомогою Mac Ports
5 $ sudo port install mc
```

А також останнім часом з'явилися бінарники під Mac OS, які можна знайти на [даній сторінці](#)⁷³. Це архів, який просто розпаковуєте і запускаєте файл, що знаходиться у ньому.

Windows - Total Commander

У Віндовсі з інсталяцією все доволі просто:

- заходите на [сторінку завантаження програми](#)⁷⁴;

⁷³<http://louise.hu/poet/midnight-commander-for-mac-os-x/>

⁷⁴<http://www.ghisler.com/download.htm>

- обираєте одне із доступних дзеркал для скачування (наприклад із [Amazon AWS⁷⁵](#));
- обираєте потрібну вам версію програми згідно вашої версії Windows та [роздрядності⁷⁶](#) (32 чи 64 біт);
- отримуєте на завантаження *.exe файл;
- запускаєте і слідуєте стандартним крокам інсталяції програми на Windows у віконному режимі та кнопочками “Далі”.

Редактор коду

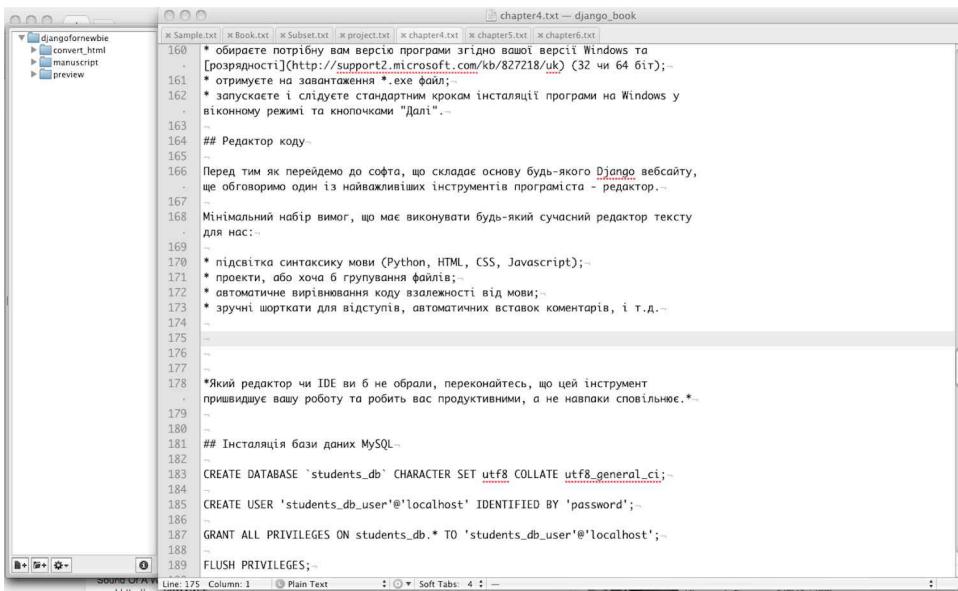
Перед тим як перейдемо до софта, що складає основу будь-якого Django вебсайту, ще обговоримо один із найважливіших інструментів програміста - редактор.

Мінімальний набір вимог, що має виконувати будь-який сучасний редактор тексту для нас:

- підсвітка синтаксику мови (Python, HTML, CSS, Javascript);
- проекти, або хоча б групування файлів;
- автоматичне вирівнювання коду взалежності від мови;
- зручні шорткати для відступів, автоматичних вставок коментарів, і т.д.

⁷⁵<http://www.ghisler.com/amazons3.php>

⁷⁶<http://support2.microsoft.com/kb/827218/uk>



The screenshot shows the TextMate code editor interface on a Mac OS X desktop. The main window displays a Python script named `chapter4.txt`. The code includes comments about installing Python and MySQL, creating a database, and creating a user. The status bar at the bottom shows "Line: 175 Column: 1 Plain Text".

```

x Sample.txt x Book.txt x Subset.txt x project.txt x chapter4.txt x chapter5.txt x chapter6.txt
160 * обираєте потрібну вам версію програми згідно вашої версії Windows та
161 [розрядності] (http://support2.microsoft.com/kb/827218/uk) (32 чи 64 біт);-
162 * отримуєте на завантаження * .exe файл;
163 * запускаєте і слідуєте стандартним крокам інсталяції програми на Windows у
164 віконному режимі та кнопочками "Далі".-
165
166 ## Редактор коду-
167
168 Перед тим як перейдемо до софта, що складає основу будь-якого Django вебсайту,
169 ще обговоримо один із найважливіших інструментів програміста - редактор.-
170
171 Мінімальний набір вимог, що має виконувати будь-який сучасний редактор тексту
172 для нас:-
173
174 * підсвітка синтаксику мови (Python, HTML, CSS, Javascript);-
175 * проекти, або хоча б групування файлів;-
176 * автоматичне вирівнювання коду взаємності від мови;-
177 * зручні шорткати для відступів, автоматичних вставок коментарів, і т.д.-
178
179 *Який редактор чи IDE ви б не обрали, переконайтесь, що цей інструмент
180 пропонує вашу роботу та робить вас продуктивніми, а не навпаки сповільнюю.-
181
182 ## Інсталяція бази даних MySQL-
183
184 CREATE DATABASE `students_db` CHARACTER SET utf8 COLLATE utf8_general_ci;-
185
186 CREATE USER 'students_db_user'@'localhost' IDENTIFIED BY 'password';-
187
188 GRANT ALL PRIVILEGES ON students_db.* TO 'students_db_user'@'localhost';-
189
190 FLUSH PRIVILEGES;-

```

Редактор коду TextMate на Mac OS

Наступні редактори задовільняють дані вимоги:

- **SublimeText⁷⁷**: платний, працює на Linux, Windows, Mac OS;
- **TextMate⁷⁸**: безкоштовний (?), працює на Mac OS;
- **Notepad++⁷⁹**: безкоштовний, працює на Windows;
- **TextWrangler⁸⁰**: безкоштовний, працює на Mac OS;
- **GEdit⁸¹**: безкоштовний, працює на Mac OS, Linux, Windows.

Але, якщо ви любите варіант “[все в одному⁸²](#)”, тоді варто також спробувати IDE - інтерактивне розробницьке середовище. Це, можна сказати, є покращеним редактором з величезним набором вбудованого функціоналу. Зазвичай “заточеного” під певний фреймворк.

⁷⁷<http://www.sublimetext.com/>

⁷⁸<http://macromates.com/>

⁷⁹<http://notepad-plus-plus.org/>

⁸⁰<https://itunes.apple.com/us/app/textwrangler/id404010395?mt=12>

⁸¹<https://wiki.gnome.org/Apps/Gedit>

⁸²<http://www.vitaliyopoda.com/2014/07/programmer-development-environment/>

Рекомендую почати пробу IDE з [PyCharm⁸³](#). Крім того, що він популярний і дійсно потужний, він ще й має вбудовану підтримку кращих практик розробки під веб-фреймворк Django.

Перед тим як зупинитись на одного редакторі чи IDE обов'язково спробуйте кілька різних варіантів, і вже тоді робіть остаточне рішення. Це дозволить вам зрозуміти, у якому редакторі ви почувастесь найкраще і працюєте най-продуктивніше.

Не важливо, що ви оберете, важливо на скільки це покращить ваш код та продуктивність.

Інсталюємо репозиторій коду Git

Я звик користуватись Git репозиторієм із командної стрічки. Важливо вміти користуватись Git з консолі особливо, коли вам приходиться працювати на віддаленому сервері, де немає графічної оболонки.

Linux Ubuntu

Інсталяція репозиторію Git на Ubuntu досить проста. В командній стрічці наберіть:

Інсталюємо Git на Ubuntu

1 \$ sudo apt-get install git-core

І це все. Тепер у вас команда git доступна в командній стрічці.

Mac OS

На Макінтоші досить просто заінсталювати git з допомогою MacPorts або Homebrew:

⁸³<https://www.jetbrains.com/pycharm/>

Інсталюємо Git на Mac OS

```
1 # Mac Ports
2 sudo port install git
3
4 # або з допомогою Homebrew
5 brew install git
```

Windows

За [даним лінком⁸⁴](#) ви автоматично почнете завантажувати Git інсталяційний файл *.exe. Після завантаження запускаєте і проходите стандартний процес встановлення софта на Віндосі. Після інсталяції в командній стрічці повинна з'явитись команда “git”.

Конфігуруємо Git

Після інсталяції Git у будь-якій із операційних систем потрібно зробити наступний мінімум налаштувань з вашої командної стрічки:

Встановлюємо глобально ваш email та ім’я для Git

```
1 $ git config --global user.name "Vitaliy Podoba"
2 $ git config --global user.email "vitaliyopodoba@gmail.com"
```

Таким чином, ваші коміти в репозиторій будуть позначені вашим іменем. Звісно, замінюєте емейл та ім’я вашими особистими даними. Немає жодних умов на ваш емейл та ім’я.

⁸⁴<http://git-scm.com/download/win>

Графічний інтерфейс

Проте, якщо на вас і так вже навалилась маса нових інструментів та систем для вивчення, пропоную також ознайомитись із Git клієнтами, що пропонують графічний інтерфейс. З їх допомогою ви з легкістю зможете працювати з Git лише використовуючи кліки миші.

Тут⁸⁵ знайдете список Git GUI клієнтів. Обираєте під свою платформу та встановлюйте. Деякі з них є безкоштовними, в той час як інші платними. Пробуйте, експериментуйте та обираєте той, який є зручнішим саме для вас.

...

Пропоную ознайомитись із [основами роботи з Git репозиторієм](#)⁸⁶ перед тим як рухатись далі.

Інсталяція бази даних MySQL

Нарешті добралися до софта, що є “серцем” веб-аплікації.

В проекті спочатку використовуватимемо sqlite3, що вже йде з мовою Python. А пізніше ми промігруємо на базу даних MySQL. Тому давайте одразу заїнсталяємо та налаштуємо MySQL на нашій розробницькій машині. Версія 5.6 і все, що вище нам підходить.

По черзі розглянемо процес на кожній із трьох операційних систем:

Linux (Ubuntu)

На Linux інсталяція MySQL досить проста, якщо користуватись пакетним менеджером aptitude та командною стрічкою Terminal:

⁸⁵<http://git-scm.com/downloads/guis>

⁸⁶<http://www.vitalypodoba.com/2014/06/git-basics/>

Інсталяція MySQL на Linux Ubuntu

```
1 # sudo - запускаємо під root користувачем, запросить від вас пароль
2 # користувача
3 $ sudo apt-get install mysql-server mysql-client libmysqlclient-dev
4
5 # в процесі інсталяції MySQL сервер запитає вас встановити пароль
6 # root користувачу; root - це головний адміністратор бази;
7
8 # тепер активуємо базу
9 $ sudo mysql_install_db
10
11 # тепер налаштовуємо видаляючи тестові бази і доступ анонімів, також
12 # встановлючи паролі доступу;
13 # ця команда запитає вас root пароль MySQL, якщо ви ще не
14 # встановлювали його, тоді залиште це поле порожнім
15 $ sudo /usr/bin/mysql_secure_installation
16
17 # Результат останньої команди буде виглядати приблизно наступним
18 # чином:
19 By default, a MySQL installation has an anonymous user, allowing
20 anyone to log into MySQL without having to have a user account
21 created for them. This is intended only for testing, and to make
22 the installation go a bit smoother. You should remove them before
23 moving into a production environment.
24
25 # видаляєте анонімів
26 Remove anonymous users? [Y/n] y
27
28 ... Success!
29
30 Normally, root should only be allowed to connect from 'localhost'.
31 This ensures that someone cannot guess at the root password from
32 the network.
33
34 # не дозволяєте root логін віддалено
```

```
35 Disallow root login remotely? [Y/n] y
36 ... Success!
37
38 By default, MySQL comes with a database named 'test' that anyone can
39 access. This is also intended only for testing, and should be
40 removed before moving into a production environment.
41
42 # видаляємо базу даних test
43 Remove test database and access to it? [Y/n] y
44 - Dropping test database...
45 ... Success!
46 - Removing privileges on test database...
47 ... Success!
48
49 Reloading the privilege tables will ensure that all changes made so
50 far will take effect immediately.
51
52 # миттєво оновляємо систему доступу до бази даних
53 Reload privilege tables now? [Y/n] y
54 ... Success!
55
56 Cleaning up...
```

Альтернативно, якщо є складнощі у використанні командної стрічки, або виникли інші проблеми на шляху, можна вручну завантажити архів *.deb із сторінки MySQL Community сервера⁸⁷.

Обирайте Ubuntu Linux у випадайці із операційними системами, а далі один із двох варіантів для завантаження:

- Ubuntu Linux [ваша версія ubuntu] (x86, 32-bit), DEB Bundle MySQL Server;
- або Ubuntu Linux [ваша версія ubuntu] (x86, 64-bit), DEB Bundle MySQL Server.

⁸⁷<http://dev.mysql.com/downloads/mysql/>

В залежності від розрядності вашої машини. Щоб дізнатись розрядність вашого процесора маючи заінстальований Linux:

```
1 uname -a
```

Ця команда виведе довгу стрічку про вашу систему. Якщо у ній знайшли “x86_64”, тоді 64 біта. Якщо ж “i386” - 32.

Запускаєте завантажений файл. Після інсталяції повторюєте в командній стрічці команди налаштування MySQL сервера:

```
1 sudo mysql_install_db  
2 sudo /usr/bin/mysql_secure_installation
```

Macintosh

Рекомендований спосіб інсталяції через бінарник.

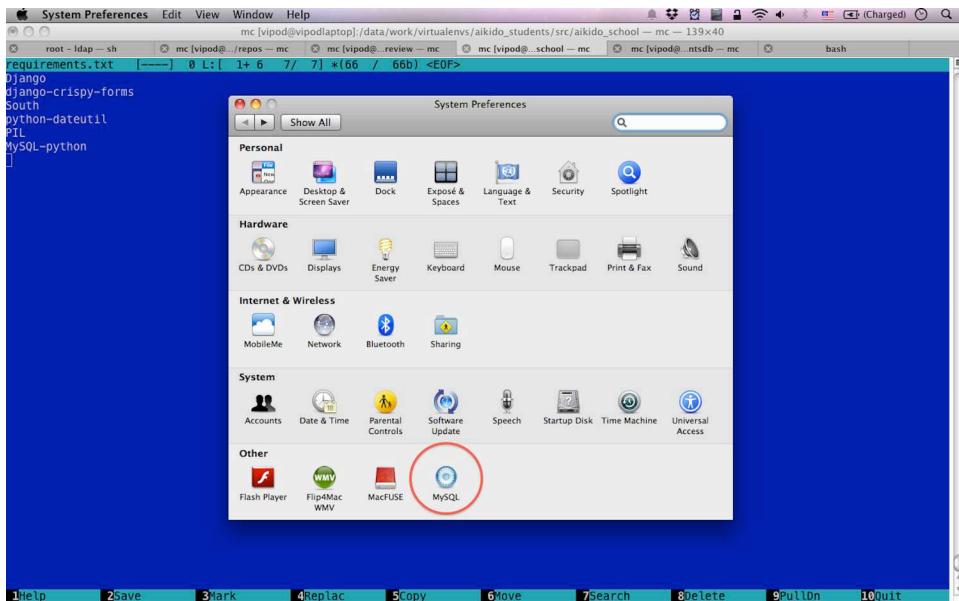
Бінарник - це спеціально підготований інсталяційний файл під необхідну операційну систему. Те, що у Windows є *.exe файлом, на Macintosh є *.dmg (disk image) файлом. Робота з обидвома типами файлів на даних операційних системах є дуже схожою.

Завантажити MySQL 5.6 інсталяційний файл для Mac OS можна на сторінці [MySQL Community сайту](#)⁸⁸. Обираєте платформу Mac OS X і в списку шукаєте “DMG Archive” саме під вашу версію Макінтоша. Отримавши *.dmg файл, відкриваєте його і отримуєте всередині два файлика, які закінчуються на *.pkg. Обидва запускаєте:

- mysql-5.6...-pkg - MySQL сервер;
- MySQLStartupItem.pkg - контрольна панель управління MySQL сервером.

⁸⁸<http://dev.mysql.com/downloads/mysql/>

Проходите стандартними інсталяційними кроками та завершуєте процес інсталяції. Підтвердженням успішного встановлення буде нова панель у ваших системних налаштуваннях під назвою MySQL:



MySQL контрольна панель в налаштуваннях

З її допомогою ви можете запускати та зупиняти MySQL сервер.

Windows

На сторінці [Download MySQL Installer⁸⁹](http://dev.mysql.com/downloads/windows/installer/5.6.html) ви також знайдете операційну систему Microsoft Windows. Завантажте *.msi або *.exe інсталятор для вашої розрядності процесора.

На даній сторінці [windows.microsoft.com⁹⁰](http://windows.microsoft.com/en-us/windows/32-bit-and-64-bit-windows) знайдете інструкцію з визначення розрядності вашого процесора на Windows.

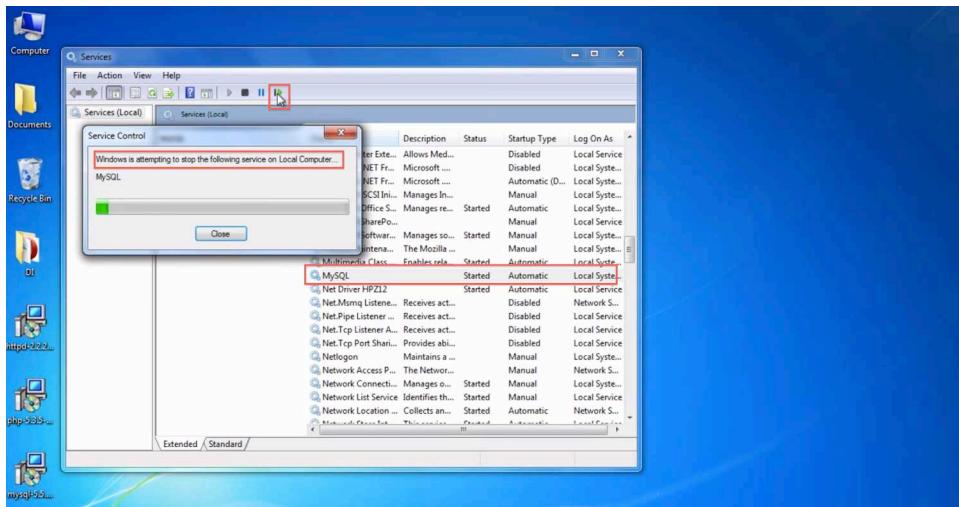
Запускаєте щойно завантажений файл і проходите усю процедуру інсталяції. При цьому варто вибрати:

⁸⁹ <http://dev.mysql.com/downloads/windows/installer/5.6.html>

⁹⁰ <http://windows.microsoft.com/en-us/windows/32-bit-and-64-bit-windows>

- Тип інсталяції: Typical (Звичайний);
- Додати MySQL bin папку до Windows Path (Шляху);
- Install As Windows Service (Зainсталювати як сервіс);
- Standard Configuration (Звичайна Конфігурація);
- Modify Security Settings (Змінити налаштування безпеки) і введіть пароль для користувача root.

Коли все зроблено зайдіть у вашу панель управління, далі Сервіси (Services), знайдіть MySQL і зробіть рестарт. На майбутнє будете цією панелькою неодноразово користуватись для запуску та зупинки MySQL сервера.



Рестарт MySQL сервісу на Windows

Конфігуруємо базу

Тепер створимо і налаштуємо нову базу даних в щойно заіnstальованому MySQL сервері. Цю базу будемо використовували далі у нашому проєкті.

В кожній з операційних систем, які ми розглянули вище, тепер має бути доступний mysql клієнт в командній стрічці. Тому запускаємо командну стрічку (Terminal або cmd або PowerShell) і запускаємо команду mysql:

Стартуємо MySQL клієнт

```
1 # якщо ви під час інсталяції MySQL не встановлювали пароль root
2 # користувачу:
3 mysql -u root
4
5 # а якщо пароль встановлено, тоді користуйтесь оцією командою:
6
7 # ця команда запитає вас пароль, який ви встановили напередодні;
8 # якщо вивід подібний на те, що ви бачите внизу, значить все добре
9 mysql -u root -p
10 # /usr/local/mysql/bin/mysql -u root -p
11 Enter password:
12 Welcome to the MySQL monitor. Commands end with ; or \g.
13 Your MySQL connection id is 7
14 Server version: 5.6.20 MySQL Community Server (GPL)
15
16 Copyright (c) 2000, 2014, Oracle and/or its affiliates. All rights
17 reserved.
18
19 Oracle is a registered trademark of Oracle Corporation and/or its
20 affiliates. Other names may be trademarks of their respective
21 owners.
22
23 Type 'help;' or '\h' for help. Type '\c' to clear the current input
24 statement.
25
26 mysql>
```

Якщо отримуєте помилки при вході в mysql, тоді перевірте:

- чи запущений ваш MySQL сервер (якщо не запущений отримаєте наступну помилку: “Can’t connect to local MySQL server through socket”);
- чи вводите правильний пароль root користувача.

Тепер у стрічці запрошення “mysql>” починаємо вводити кілька SQL команд:

Створюємо і налаштовуємо нову базу MySQL

```
1 # створюємо таблицю студентів students_db; кодування utf8;
2 mysql> CREATE DATABASE `students_db` CHARACTER SET utf8 COLLATE utf8\
3 _general_ci;
4
5 # створюємо користувача students_db_user, замість password вставте
6 # свій пароль і запам'ятайте його; він нам пізніше пригодиться;
7 mysql> CREATE USER 'students_db_user'@'localhost' IDENTIFIED BY 'pas\
8 sword';
9
10 # надаємо усіх прав нашому користувачу над нашою новоствореною базою;
11 mysql> GRANT ALL PRIVILEGES ON students_db.* TO 'students_db_user'@'\
12 localhost';
13
14 # рестартуємо дозволи для негайногго застосування наших налаштувань по
15 # користувачу;
16 mysql> FLUSH PRIVILEGES;
17
18 # виходимо
19 mysql> quit
```

Усі бекслеші ("\"), що ви бачите у прикладах коду та командної стрічки при переносі тексту на новий рядок, були додані автоматично системою верстки книги. Їх переносити не потрібно при використанні коду у ваших експериментах.

Інсталяція Python

От і добрались до інсталяції інтерпретатора Python!

В нашому проекті ми використовуватимемо Python 2.7. Це остання версія на даний момент в серії Python 2.x. Python 3.x в даній книзі не будемо розглядати. Чому? В даній статті я відповідаю на це запитання: “[Python 2 vs Python 3⁹¹](#)”.

Загалом я планую написати додаток, або просто курс після книги у якому продемострую процес міграції проекту даної книги на Python 3. Але це вже буде після того як ви освоїти основи Django на Python 2.

Linux (Ubuntu) та Macintosh

На Linux (Ubuntu) та старіших версіях Mac OS Python уже є вбудований в початкову інсталяцію операційної системи.

Щоб перевірити чи є Python у вас на Linux та Mac OS, а також якої він версії, відкриваємо командну стрічку і пробуємо:

Перевіряємо Python на Mac OS і Linux (Ubuntu)

```
1 # просто набираємо python, якщо немає бінарника, отримаєте наступну
2 # помилку
3 $ python
4 -sh: python: command not found
5
6 # інакше отримаєте щось подібне на наступне
7 $ python
8 Python 2.7.3 (default, Jan  4 2014, 00:47:50)
9 [GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.9)] on \
10 darwin
11 Type "help", "copyright", "credits" or "license" for more informatio\
12 n.
13 >>>
14 # використовуйте Cmd-D або Cmd-C клавіши, щоб вийти з Python
```

⁹¹<http://www.vitaliyopoda.com/2014/05/python2-or-python3/>

```
15 # інтерпретатора
16
17 # таким чином також можна перевірити версію Python
18 $ python -V
19 Python 2.7.3
20
21 # як побачити де знаходиться бінарник інтерпретатора
22 $ which python
23 /usr/bin/python
24
25 # а таким чином можна глянути де лежать бібліотеки Python
26 $ python
27 Python 2.7.3 (default, Jan  4 2014, 00:47:50)
28 [GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.9)] on \
29 darwin
30 Type "help", "copyright", "credits" or "license" for more informatio\
31 n.
32 >>> import os
33 >>> os
34 <module 'os' from '/usr/local/Cellar/python/2.7.3/Frameworks/Python.\
35 framework/Versions/2.7/lib/python2.7/os.pyc'>
36 >>>
```

Решіткою (#) я позначатиму коментарі у прикладах із кодом.

Якщо версія Python виявилась в серії Python 2.7.x, тоді в принципі можна починати працювати із ним без додаткових інсталяцій.

Якщо немає Python в системі взагалі, тоді на Linux (Ubuntu) інсталюєте усе наступними командами:

Інсталяція Python на Linux (Ubuntu) та усіх залежностей

```
1 $ sudo apt-get install build-essential make cmake scons
2 $ sudo apt-get install autoconf automake autoconf-archive
3 $ sudo apt-get install gettext libtool
4 $ sudo apt-get install libbz2-dev zlib1g-dev
5 $ sudo apt-get install libexpat-dev libncurses-dev
6 $ sudo apt-get install libjpeg62-dev libcurl4-openssl-dev
7 $ sudo apt-get install libssl-dev python python-dev curl
8
9 # build-essential - gcc компілятор С коду;
10 # make, cmake, scons - бібліотеки побудови бінарних пакетів в Linux;
11 # conf пакети - утиліти ефективного створення конфігурацій;
12 # gettext - бібліотека перекладів;
13 # libbz2-dev - хідера для архіватора;
14 # libtool - інструменти по роботі із shared бібліотеками;
15 # libssl-dev - розробницькі хідера для ssl підтримки в python;
16 # libexpat-dev - розробницькі хідера для xml парсера;
17 # libncurses-dev - хідера бібліотеки для побудови
18 # термінальних GUI;
19 # python - сам python;
20 # curl - утиліта, з допомогою якої далі під час інсталяції ми будемо
21 # отримувати дані з інтернету в командній стрічці;
22 #
```

Якщо ви отримали повідомлення типу “Not Found Package” (пакет не знайдено) при запуску однієї із вище наведених команд, тоді спробуйте спочатку оновити ваш індекс репозиторіїв на Ubuntu: “\$ sudo apt-get update;sudo apt-get upgrade”. Після цього повторіть команди, які поламались напередодні.

А на новіших системах Mac OS Python йде як частина XCode середовища, яке

ви можете заінсталювати з [Apple Store⁹²](#).

Таким чином ви отримаєте дефолтну (останню стабільну версію підготовану для вашої системи) версію Python.

Тим не менше, навіть якщо уже глобально встановлений Python і він потрібної версії, рекомендую продовжити до наступної секції та встановити його ще один раз через технологію zc.buildout.

Інсталяція Python з buildout на Ubuntu та Mac OS

Цей метод працює і на Linux (Ubuntu) і на Mac OS. Для чого він потрібний, особливо якщо вже є Python потрібної версії?

Цей метод дозволяє:

- за вас поставити усі необхідні бібліотеки, на які залежить Python;
- поставити одночасно кілька Python версій і все це одним махом.

[zc.buildout⁹³](#) - це інструмент для управління розробницькими середовищами. Він являється Python пакетом і для його користування нам все одно пригодиться попередньо заінсталюваний Python.

Для інтерпретатора Python в спільноті Zope, Plone CMS програмістів створили спеціальну конфігурацію білдаута: [buildout.python⁹⁴](#). Ця конфігація якраз і дає можливість кількома простими кроками поставити одразу кілька локальних версій Python із сурсів.

Перед наступними кроками переконайтесь, що ви виконали інструкції із попередньої секції та маєте довільну версію мови Python. Адже сам інструмент zc.buildout написаний на даній мові і без нього не запуститься.

Вся робота над проектом проходитиме в папці /data в корені файлової системи. Щоб створити дану папку на Лінуксі потрібно права рутового користувача (тобто адміністратора). Після того як створимо її, передамо права на володіння

⁹²<https://developer.apple.com/xcode/>

⁹³<https://pypi.python.org/pypi/zc.buildout/2.2.3>

⁹⁴<https://github.com/collective/buildout.python>

вашому власному користувачеві. І вже надалі користуватиметься лише вашим власним користувачем для інсталяції робочого середовища Django та роботи над кодом проекту.

Готуємо головну папку data

```
1 # створюємо кореневу папку data в корені файлової системи
2 # команда sudo запускає передану їй команду під адміном,
3 # тому вас попросять ввести ваш власний пароль
4 $ sudo mkdir -p /data
5
6 # передаємо усі права на папку вашому користувачеві
7 # у моєму випадку це користувач vipod, підставте власного;
8 # маєте замінити стрічку vipod у двох місцях: до двокрапки
9 # і після
10 $ sudo chown -R vipod:vipod /data
```

Тепер ви зможете виконувати усі наступні дії всередині папки /data під вашим користувачем. Увага! Не використовуйте команду sudo, якщо не впевнені, що це необхідно. Особливо, якщо в книзі наведена команда без приставки sudo.

Альтернативно можете створити папку для проекту у домашній папці вашого користувача і там продовжувати усю роботу. При цьому не забувайте змінювати шляхи наведені у книзі і підпасовувати під свою структуру папок та файлів. У Лінуксі та Макінтоші домашня папка користувача альтернативно позначається знаком тильди (~).

Відкриваємо командну стрічку і починаємо запускати наш білдавт:

Налаштування Python Buildout

```
1 # у мене усі білдаєти лежать у /data/work/buildouts;
2
3 # рекурсивно створюємо необхідну структуру папок
4 # (проте ви можете собі придумати власну структуру);
5 $ mkdir -p /data/work/buildouts;
6
7 # заходимо у щойно створену папочку buildouts;
8 $ cd /data/work/buildouts
9
10 # клонуємо python buildout репозиторій; для цього лінк копіюю із
11 # поля SSH clone URL на сторінці репозиторію в github.com:
12 $ git clone https://github.com/collective/buildout.python.git python
13
14 # заходимо у щойно склонований репозиторій
15 $ cd python
```

Якщо при клонуванні репозиторію buildout.python вам видає, що у вас немає достатньо прав (Permission Denied), тоді клонуйте даний репозиторій із наступної адреси, яка є “readonly” (тобто лише на вичитування даних, але не на запис): <https://github.com/collective/buildout.python.git>

Не використовуйте sudo із жодною із вище наведених команд. Усі вони повинні запускатись під вашим звичайним користувачем. sudo зробить папки підвласними root користувачу, що призведе до подальших проблем із роботою над проектом під вашим звичайним користувачем.

Також рекомендую ознайомитись базово із [системою дозволів⁹⁵](#) в операційні

⁹⁵<http://bit.ly/vplinuxsecurity>

системі Linux.

Перед запуском наступної команди ви можете поредагувати файл buildout.cfg в корені білдаута. Основний параметр - це опція parts, яка має список додаткових бібліотек (readline, libjpeg), а також список версій Python, які ми хочемо заінсталювати. Так, можна одразу мати кілька локальних версій Python інтерпретатора заінсталюваного на своєму комп'ютері.

Я закоментую усі версії крім потрібної нам python27:

buildout.cfg файл, опція parts

```
1 parts =
2     ${buildout:base-parts}
3     ${buildout:readline-parts}
4     ${buildout:libjpeg-parts}
5     # ${buildout:python24-parts}
6     # ${buildout:python25-parts}
7     # ${buildout:python26-parts}
8     ${buildout:python27-parts}
9     # ${buildout:python32-parts}
10    # ${buildout:python33-parts}
11    # ${buildout:python34-parts}
12    ${buildout:pypy-parts}
13    ${buildout:pypy3-parts}
14    ${buildout:links-parts}
```

Зберігаємо зміни у файлі buildout.cfg і рухаємось далі:

Запуск білдавта

```
1 # нашим напередодні заінстальованім Пітоном запускаємо bootstrap.py
2 $ python bootstrap.py
3
4 # скрипт bootstrap.py збере для нас сам пакет zc.buildout та
5 # необхідний мінімум пакетів для запуску білдавта
6
7 # також після запуску bootstrap.py матимемо новий скрипт у підпапці
8 # bin: buildout; запускаємо його для старту білдавта і збірки уього
9 # середовища:
10 $ ./bin/buildout
11
12 # ось так виглядатиме процес роботи buildout
13 Getting distribution for 'buildout.extensionscripts'.
14 zip_safe flag not set; analyzing archive contents...
15 Got buildout.extensionscripts 1.0.
16 Getting distribution for 'plone.recipe.command'.
17 Got plone.recipe.command 1.1.
18 Getting distribution for 'hexagonit.recipe.download'.
19 Got hexagonit.recipe.download 1.7.
20 Getting distribution for 'collective.recipe.cmmi==0.5'.
21 Got collective.recipe.cmmi 0.5.
22 Getting distribution for 'zc.recipe.cmmi<=1.4.0'.
23 Got zc.recipe.cmmi 1.3.6.
24 Getting distribution for 'collective.recipe.template'.
25 Got collective.recipe.template 1.11.
26 Getting distribution for 'z3c.recipe.runscript'.
27 Got z3c.recipe.runscript 0.1.3.
28 Unused options for buildout: 'python24-parts' 'python33-parts' 'pyth\
29 on24-pil-install-args' 'python26-parts' 'python32-parts' 'python25-p\
30 il-install-args' 'python34-parts' 'python25-parts'.
31 ...
```

Процес компіляції та конфігурації Python із вихідного коду може зайняти кілька хвилин, тому можна зробити паузу і випити чаю.

Якщо процес закінчився без помилок, то ви повинні побачити щось подібне до цього у своїй командній стрічці:

Успішно завершений запуск buildout

```
1 -----  
2 --- TKINTER support available  
3 --- JPEG support available  
4 *** OPENJPEG (JPEG2000) support not available  
5 --- ZLIB (PNG/ZIP) support available  
6 --- LIBTIFF support available  
7 --- FREETYPE2 support available  
8 *** LITTLECMS2 support not available  
9 *** WEBP support not available  
10 *** WEBPMUX support not available  
11 -----  
12 To add a missing option, make sure you have the required  
13 library, and set the corresponding ROOT variable in the  
14 setup.py script.  
15  
16 To check the build, run the selftest.py script.  
17  
18 Adding Pillow 2.6.1 to easy-install.pth file  
19 Installing pilconvert.py script to /data/work/buildouts/buildout.pyt\  
20 hon/parts/opt/bin  
21 Installing pildriver.py script to /data/work/buildouts/buildout.pyt\  
22 on/parts/opt/bin  
23 Installing pilfile.py script to /data/work/buildouts/buildout.python\  
24 /parts/opt/bin  
25 Installing pilfont.py script to /data/work/buildouts/buildout.python\  
26 /parts/opt/bin  
27 Installing pilprint.py script to /data/work/buildouts/buildout.python\  
28 n/parts/opt/bin  
29  
30 Installed /data/work/buildouts/buildout.python/partsopt/lib/python2\  
31 .7/site-packages/Pillow-2.6.1-py2.7-macosx-10.6-x86_64.egg
```

```
32 Processing dependencies for Pillow
33 Finished processing dependencies for Pillow
34 Unused options for python-2.7-PIL: 'update-command'.
35 Installing python-2.7-test.
36 Unused options for python-2.7-test: 'update-script'.
37 Installing install-links.
```

Ще одним доказом правильно закінченого білдавт процесу буде наявність наступних скриптів у вашій bin директорії білдавта:

- buildout
- install-links
- virtualenv-2.7

Якщо бракує virtualenv-2.7, тоді швидше за все стала помилка, яку ви можете бачити наприкінці вікна вашої командної стрічки. Пробуйте розібратись. Якщо не виходить - звертайтесь по допомогу у закриту групу підтримки.

Так, python бінарника немає у підпапці bin нашого білдавта. Згідно кращих практик рекомендують не використовувати напряму заінсталюваний Python, а використовувати натомість лише віртуальні середовища (virtualenv). Більше про це далі.

Windows

Якщо ви все ж таки вирішили спробувати практикуватись над проектом працюючи на Windows, тоді тримайте опис інсталяції Python та Django на моєму блозі: [Як заінсталювати Python 2, Pip, Virtualenv i Django на Windows?](#)⁹⁶.

⁹⁶<http://www.vitaliy-podoba.com/2014/09/how-to-install-python2-pip-virtualenv-django-windows/>

Там виникають кілька нюансів під час інсталяції. Якщо у вас будуть ще й інші проблеми в процесі, тоді звертайтесь у закриту групу підтримки по допомогу. Windows - конячка вибаглива ;-)

Інсталяція virtualenv та Django

Маючи Python тепер дуже легко закінчити наш процес налаштування Django. Інсталювати Django ми не будемо напряму в Python, а у окремо створене так зване віртуальне середовище (пакет [virtualenv⁹⁷](#)).

Віртуальне середовище - це можливість мати кілька Python проектів і працювати над ними паралельно. При цьому один іншому не буде перешкоджати. Це окремі папки із своєю копією Python та усіх заінсталюваних додаткових бібліотек.

Розглянемо у окремих під-секціях інсталяцію віртуального середовища для обох випадків: з глобальним Python і з Python заінсталюваним з допомогою білдаута.

Глобальний Python

Якщо ви вирішили зупинитись на глобально заінсталюваному інтерпретаторі Python, тоді швидше за все вам ще потрібно поставити [pip⁹⁸](#) (менеджер пакетів в Python), а тоді з його допомогою virtualenv:

⁹⁷<http://virtualenv.readthedocs.org/en/latest/>

⁹⁸<https://pypi.python.org/pypi/pip>

Інсталюємо pip і virtualenv в глобальний Python

```
1 # завантажуєте https://bootstrap.pypa.io/get-pip.py
2 # скрипт будь-яким зручним для вас способом
3
4 # ось варіант завантаження через команду в консолі:
5 wget https://bootstrap.pypa.io/get-pip.py
6
7 # робимо під рутом (адміном), команда sudo попросить вашого паролю;
8 # запускаємо Python-ом щойно завантажений скрипт
9 $ sudo python get-pip.py
10
11 # після попередньої команди у вас повинна з'явитись глобально
12 # доступна утиліта 'pip', ії ми використаємо для інсталяції virtualenv
13 $ sudo pip install virtualenv
```

Тепер маємо глобально доступний в командній стрічці скрипт “virtualenv”.

Python в zc.buildout

Якщо ж ви інсталювали Python через білдавт, тоді у вас уже є заінсталюваний скрипт віртуального середовища і знаходиться він у підпапці bin у білдавті.

Відповідно нічого інсталювати не потрібно при використанні Python із білдавта, ми просто запустимо скрипт “bin/virtualenv-2.7”, що знаходиться в білдавті.

Віртуальне середовище

У мене на власному комп’ютері є багато проектів на мові Python і тому я створив окрему папку для усіх Python віртуальних середовищ, між якими переключаюсь при потребі. Рекомендую вам також обзавестись окремою папкою для Python проектів:

Створюємо віртуальне середовище

```
1 # створюємо рекурсивно папочку для наших віртуальних середовищ, у
2 # мене це:
3 $ mkdir -p /data/work/virtualenvs
4
5 # заходимо у щойно створену папку
6 $ cd /data/work/virtualenvs
7
8 # створюємо віртуальне середовище для нашого проекту studentsdb
9 # використовуючи глобальний Python --no-site-packages: не копіюємо
10 # додаткових пакетів з основного Python-a
11 $ virtualenv studentsdb --no-site-packages
12
13 # а це варіант, якщо створюємо віртуальне середовище використовуючи
14 # наш Python з білдаута, що ми створили у попередній секції:
15 $ /data/work/buildouts/python/bin/virtualenv-2.7 studentsdb --no-sit\
16 e-packages
17
18 # результатом попередньої команди буде створена підпапка studentsdb з
19 # копією Python та pip; заходимо у середовище та активовуємо його
20 $ cd studentsdb
21
22 # скрипт activate встановить кілька шляхів і змінних у середовище
23 # командної стрічки, що дозволить нам використовувати команди
24 # python, pip та інші із підпапочки bin віртуального середовища без
25 # вказування повного шляху до них
26 $ source bin/activate
27
28 # ось так виглядатиме стрічка запрошення у командній стрічці після
29 # запуску скрипту activate
30 (studentsdb)$
31
32 # тобто ми почали працювати в контексті даного віртуального
33 # середовище, і якщо глянемо звідки тепер беруть наші скрипти, то
34 # переконайємось у цьому:
```

```
35 $ which python
36 (studentsdb)$ which python
37 /data/work/virtualenvs/studentsdb/bin/python
38 (studentsdb)$ which pip
39 /data/work/virtualenvs/studentsdb/bin/pip
```

У віртуальному середовищі маємо наступні важливі для нас папки:

- bin: бінарники (python, pip, activate, тут також будуть скоро django скрипти);
- lib: пітонівські пакети, сюди будуть також заінсталені пакети Django (підпапка python2.7/site-packages), сюди ми не раз будемо заглядати під час розробки.

Врешті решт ставимо Django:

Встановлення Django через pip

```
1 (studentsdb)$ pip install Django
```

Вище наведена команда знайде в неті пакет [Django⁹⁹](#), завантажить його, розпакує та поставить у наше віртуальне середовище. Доказом успішно заіnstальованого Django буде скрипт django-admin.py всередині підпапки bin нашого віртуального середовища.

Django Проект

Маючи Django фреймворк встановлений у нашему віртуальному середовищі, можемо тепер створити наш перший проект, запустити його та переглянути сторінку привітання у браузери.

Я зазвичай створюю підпапочку ‘src’ в корені віртуального середовища, куди кладу весь власний код включно з python пакетами, Django проектами та аплікаціями:

⁹⁹<https://pypi.python.org/pypi/Django>

Створення Django проекту

```
1 # переконайтесь, що знаходитесь в корені віртуального середовища
2 $ pwd
3 /data/work/virtualenvs/studentsdb
4
5 # створюємо підпапку src і заходимо в неї:
6 $ mkdir src
7 $ cd src
8
9 # скористуємося скриптом django-admin, який з'явився у нас в
10 # підпапці bin нашого віртуального середовища після того, як ми
11 # заінсталювали Django;
12
13 # команда startproject створить для нас Django проект
14 $ ../bin/django-admin startproject studentsdb
```

Команда django-admin (а також скрипт manage.py) мають цілий набір підготовлених команд, які дозволяють працювати із базою даних, запускати міграції, запускати розробницький Django сервер і ще дуже багато інших речей. Для ознайомлення із повним списком використовуйте опцію “`-help`”: `“django-admin.py -help”`.

Django Проект - це заготований для нас набір папок та файлів (а в поняттях Python - пакетів та модулів), які складають кістяк веб-аплікації в Django. Проект дає нам хороший старт для написання нашого власного коду і наперед задає набір правил що і куди має йти.

Після створення нашого середовища матимемо наступну структуру файлів і папок:

- `manage.py` - аналог `django-admin`, але в контексті проекту;
- `studentsdb` - папка Django проекту;

- `__init__.py` - порожній модуль, індикатор Python пакету;
- `settings.py` - модуль із налаштуваннями Django проекту;
- `urls.py` - налаштування URL диспетчера;
- `wsgi.py` - модуль, що робить нашу аплікацію WSGI аплікацією.

Структуру проекту ми розбиралимо детальніше в наступних главах книги.

Старт Проекту

Спочатку поредагуємо налаштування нашого проекту, які лежать в модулі `settings.py` і перенесемо файл `sqlite3` бази даних на рівень вище, за межі Django проекту. В наступній секції ми додамо наш проект у Git репозиторій. Не рекомендується тримати генеровані файли та файли баз даних у репозиторії коду. Таким чином ми гарантуємо, що файл бази даних не попаде в репозиторій:

settings.py

```
1 DATABASES = {  
2     'default': {  
3         'ENGINE': 'django.db.backends.sqlite3',  
4         'NAME': os.path.join(BASE_DIR, '..', 'db.sqlite3'),  
5     }  
6 }
```

В змінній `DATABASES` ми змінили шлях до `db.sqlite3` файла на рівень вище додавши “..” одразу за “`BASE_DIR`” змінною. Рядочок 4 у вище наведеному прикладі коду.

Спробуйте розібратись як калькулюється `BASE_DIR`, а потім ‘`NAME`’ в `DATABASES` самостійно.

Перед запуском сайту нам ще потрібно початково налаштувати базу даних. Робиться це доволі просто командою скрипта `manage.py`:

Налаштовуємо базу даних

```
1 # ця команда створить усі необхідні таблиці у нашій базі згідно
2 # списку активованих аплікацій всередині модуля settings.py
3
4 # ця команда також запросятиме у вас чи створювати адміна, дайте
5 # відповідь так та введіть нік адміністратора, пароль та його
6 # емейл адресу
7 (studentsdb)$ python manage.py syncdb
8 Operations to perform:
9   Apply all migrations: admin, contenttypes, auth, sessions
10 Running migrations:
11   Applying contenttypes.0001_initial... OK
12   Applying auth.0001_initial... OK
13   Applying admin.0001_initial... OK
14   Applying sessions.0001_initial... OK
15
16 You have installed Django's auth system, and don't have any superuse\
17 rs defined.
18 Would you like to create one now? (yes/no): yes
19 Username (leave blank to use 'vipod'): admin
20 Email address: admin@admin.com
21 Password: *****
22 Password (again): *****
23 Superuser created successfully.
```

Після вище запущеної команди у нас з'явиться файлик бази даних за адресою:
/data/work/virtualenvs/studentsdb/src/db.sqlite3

Ну і нарешті запуск сервера:

Старт Django проекту

```
1 (studentsdb)$ python manage.py runserver
2 Performing system checks...
3
4 System check identified no issues (0 silenced).
5 November 01, 2014 - 13:01:39
6 Django version 1.7.1, using settings 'studentsdb.settings'
7 Starting development server at http://127.0.0.1:8000/
8 Quit the server with CONTROL-C.
```

Таким чином runserver повідомив нас, що сервер успішно запущено під адресою <http://127.0.0.1:8000>. Відкриваємо її у своєму веб-переглядачі і маємо повідомлення про успішний запуск нашої аплікації:



Демо сторінка новоствореного Django проекта

Щоб вийти із даного серверного процесу, потрібно скористатись набором клавіш Ctrl-C.

В папці /data/work/virtualenvs/studentsdb/src/studentsdb лежатиме наш власний код по проекту і там ми проводитимемо більшість свого часу.

Тепер, маючи базову структуру проекту, можемо переходити до наступного кроку.

Git

На завершення глави маємо закомітити щойно створений проект у репозиторій коду. У репозиторій покладемо корінь нашого Django проекту:

Django проект в Git

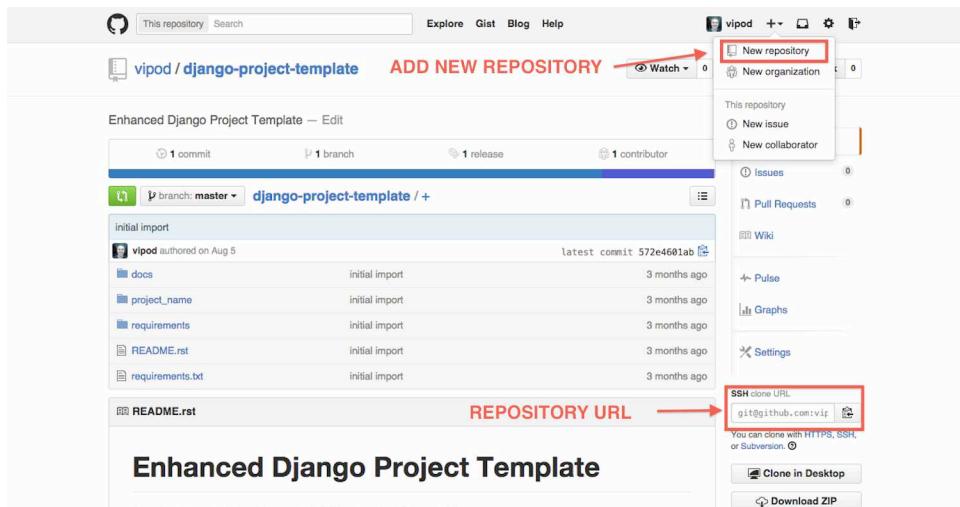
```
1 # переконаємось, що ми в корені нашого проекту
2 $ pwd
3 /data/work/virtualenvs/studentsdb/src/studentsdb
4
5 # ініціалізуємо новий репозиторій
6 $ git init
7
8 # додаємо усі файли до індекса git, переконайтесь що у вас немає
9 # *.pyc файлів в проекті, а ще краще глобально налаштуйте ігнор
10 # "поганим" файлам
11 $ git add *
12
13 # комітимо усі файли в локальний репозиторій, не забудьте додати
14 # змістовний коментар до вашого коміту
15 $ git commit -a -m 'initial project template'
```

Ось інструкція¹⁰⁰ як налаштовувати глобальний файл із списком ігнорованих Git-ом файлів. Мінімальний набір шаблонів для ігнору в Python: *.pyc, *~, *.egg-info, *.mo, *.egg.

Також рекомендую зареєструватись на github.com¹⁰¹ та закинути свій код туди у власний новостворений публічний репозиторій:

¹⁰⁰<http://habrahabr.ru/post/202696/>

¹⁰¹[https://github.com/](https://github.com)



Створіть репозиторій на github.com та скопіюйте його адресу

Також, щоб мати можливість не лише клонувати репозиторії із github.com, але й закидати туди свої оновлення, потрібно згенерувати свій SSH ключ та закинути публічну його частину у свій екаунт на github.com. [Тут](#)¹⁰² ви можете знайти детальну інструкцію як це можна зробити для операційних систем Windows та Linux. Зробіть це зараз, щоб мати можливість продовжити до виконання наступних кроків описаних нижче.

Маючи адресу новоствореного репозиторію запуште свій локальний репозиторій туди:

```

1 # додаємо новостворений репозиторій як remote (віддалений
2 # репозиторій) до вашого локального репозиторію, під іменем origin;
3
4 # тут має бути адреса до вашого власного репозиторію:
5 $ git remote add origin git@github.com:vipod/test.git
6
7 # синхронізуєте (закидуєте) ваш локальний репозиторій із
8 # новоствореним; пушим зміни у віддалений репозиторій origin в
9 # дефолтну гілку master:
10 $ git push origin master

```

¹⁰²<http://webhamster.ru/site/page/index/articles/comp/171>

Тепер перевірте чи сторінка вашого репозиторію на github.com оновилась вашими змінами.

Домашнє завдання

В даній главі я не намагався розповісти 100% ситуацій і шляхів інсталяції програм. На кожній операційній системі є свої проблеми, які залежать від різних версій пакетів, розрядності процесора, набору поставлених бібліотек. Я вже не згадую Windows, який найчастіше використовується у не ліцензійному вигляді.

Я навів еталонний варіант встановлення софта, коли все йде по плану. Тому, якщо у вас виникають труднощі (а швидше за все так і буде), звертайтесь у групу підтрики і будемо разом шукати розв'язку. Дана глава книги, як ніяка інша потребуватиме регулярних оновлень та покращень. Думаю разом ми знайдемо найскладніші місця у встановленні робочого середовища та детальніше опишемо розв'язки.

Думаю для вас і без того це був непростий розділ. Дехто із моїх студентів працював близько тижня, а деколи і більше, щоб налаштувати все як треба. Перший блін комом, як то кажуть. Тому наберіться терпіння. Прийдеться пробувати багато різних нових речей і кількома різноманітними способами. У мене до тепер час до часу виникають проблеми із налаштуванням тих чи інших інструментів, тому дані навички вам пригодяться і в майбутньому.

...

Таким чином наприкінці даної глави у вас має бути:

- в ідеалі операційна система Linux, дистрибутив Ubuntu; або хоча б Mac OS;
- база даних MySQL встановлена та відконфігуркована;
- Python 2.7.x із virtualenv та pip;
- створені: віртуальне середовище, Django 1.7;
- ваш Django проект на github.com.

Також на домашнє завдання:

- освоїти основні операції з файлами і папками в командній стрічці
- спробувати кілька редакторів тексту та IDE і обрати один для себе;
- при потребі обрати додатковий менеджер файлів;

І коли все готово - опублікуйте в закритій групі підтримки повідомлення про обране вами робоче середовище, редактори і т.д.

Я переконаний, що в процесі налаштування вашого робочого середовища, починаючи від операційної системи і закінчуячи запуском Django проекту, у вас виникала сила силенна проблем, помилок та незрозумілих кроків на шляху. А можливо навіть деякі із них ще й досі залишились нерозв'язаними повністю. В такому випадку рекомендую зробити дві наступні речі:

- завітайте до закритої групи підтримки та пошукайте за подібними проблемами, можливо знайдете відповідь на свої питання; якщо ні, тоді запостіть свої проблеми і думаю вам там допоможуть;
- з'їжджайте нарешті із Віндовса ;-)

Допоки все не запрацює, будь-ласка, не переходить до наступної глави книги.

...

У наступній главі почнемо трохи кодити. Але ще не на мові Python, а на HTML та CSS. Оглянемо вигляд головної сторінки нашої аплікації - список студентів та повністю її зверстаємо використовуючи HTML, CSS та фреймворк Twitter Bootstrap. В наступній главі ми спробуємо розібраться із базисом веб верстки.

5. Верстаємо головну сторінку: лейаут та список студентів

У цій главі ми вперше активно попрацюємо у нашому редакторі коду. Тому, якщо ви ще досі не визначились із ним, будь-ласка, верніться до попередньої глави, та оберіть, хоча б для тимчасового використання, редактор або IDE.

Займемось статичною версткою головної сторінки нашої аплікації. А саме, напишемо необхідний HTML і CSS код для сторінки із списком студентів:

The screenshot shows a web application titled "Сервіс Обліку Студентів". At the top, there is a dropdown menu labeled "Група: Усі Студенти". Below the header, there are three navigation tabs: "Студенти" (selected), "Відвідування", and "Групи". A success message "Студент успішно доданий." is displayed in a yellow box. The main section is titled "База Студентів" and contains a table with student data. The columns are: "#", "Фото", "Прізвище ↑", "Ім'я", "№ Білету", and "Дії". The table rows show three students: 1. Корост Андрій (2123), 2. Подоба Віталій (254), and 3. Притула Тарас (5332). On the right side of the table, there is a context menu with options: "Редагувати", "Відвідування", and "Видалити". At the bottom left, there is a copyright notice: "© 2014 Сервіс Обліку Студентів".

Сторінка із списком студентів

По-суті, ми перенесемо дане зображення у наш веб-переглядач.

Перед продовженням також хочу поцікавитись як ви справились із домашнім завданням, де я просив вас пройти кілька туторіалів та курсів по HTML та CSS?

Дані теоретичні знання вам дуже пригодяться у цій главі, адже ми не будемо розбирати основи синтаксису даних мов, а вже використовувати конкретні теги та правила.

Якщо вам буде незрозумілий контекст коду із кусків наведених у прикладах до даної глави, тоді звертайтесь до коду, що йде разом із книгою. Там ви зможете побачити кінцевий результат, що допоможе вам краще розуміти окремі куски наведені далі у прикладах.

Інструментарій

Першим ділом давайте розберемось із тими інструментами, що пригодяться нам під час верстки.

Редактор ми уже згадали. Мінімальні вимоги до нього - це підсвідка HTML, CSS та Javascript коду. Бажано також мати проекти (групування файлів) та кілька автоматичних дій як коментування коду та авто-відступи. Я використовую TextMate на Mac OS.

Firefox

Протягом книги для фронт-енд робіт та тестів я використовуватиму браузер Firefox останньої версії. Вам не обов'язково користуватись саме ним. Дуже хорошиою альтернативою є Chrome. Верстка у обидвох цих браузерах зазвичай працює подібно і якщо ви зверстали сторінку під один із цих браузерів, вона без додаткових змін буде працювати у іншому.

Не рекомендую починати верстати одразу в Internet Explorer, адже там є більше нюансів. Зазвичай ми у клієнтських проектах верстаемо спочатку під Chrome та Firefox. А вже коли робота готова, тестиємо на Internet Explorer, Safari та Опера (звичайно якщо є в цьому потреба). В проекті даної книги ми не будемо займатись крос-браузерною чи крос-платформенною підтримкою верстки. Лише браузер Firefox. Додаткові бібліотеки jQuery та Twitter Bootstrap дозволять нам не надто хвилюватись про це.

Завантажити та заінсталювати веб-переглядач Firefox ви можете [тут¹⁰³](#). Ця сторінка автоматично визначить вашу операційну систему та дасть вам на завантаження правильний файл.

Firebug

Я вже згадував цей інструмент в попередніх главах. Якщо ви його ще не заінсталювали, будь-ласка, зробіть це зараз у випадку, якщо ви плануєте користуватись Firefox браузером під час верстки.

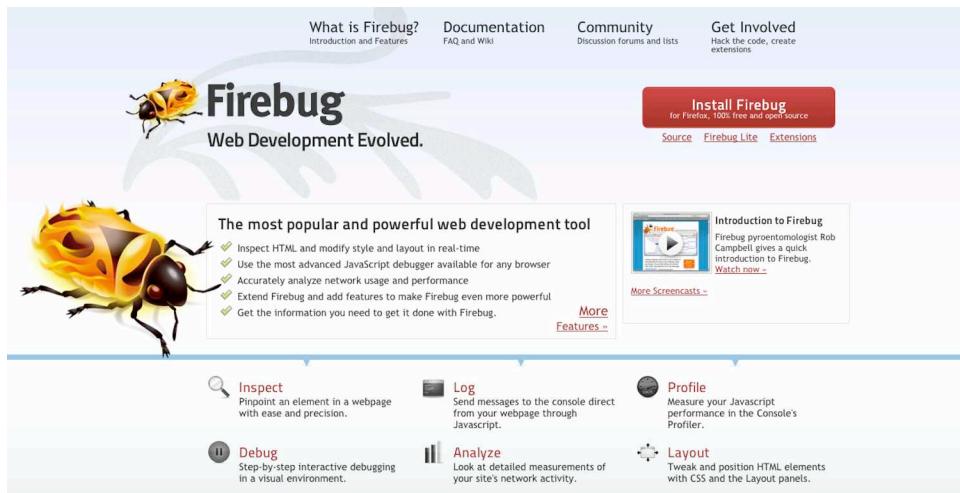
Після інсталяції браузера Firefox відкрийте ним сторінку плагіна [Firebug¹⁰⁴](#), натисніть кнопку “Install Firebug” і завершіть інсталяцію. В старіших версіях Firefox необхідно було рестартувати браузер для повного встановлення плагіна, але в останніх версіях цього не потрібно робити.

Загалом кожен сучасний браузер має уже свій власний інструмент розробки, який дуже спрощує дебаг фронт-енд коду, як HTML, так і CSS та Javascript. Достатньо клікнути на сторінці правою клавішею миші та обрати пункт меню “Inspect Element” і ви відкриєте вікно дебаг панелі.

Нажаль передати увесь процес дебагу і користування такими інструментами як Firebug досить важко в текстовому режимі. Але я непогано демонструю процес дебагу та відладки коду у відео уроках, що йдуть із Рекомендованим пакетом книги.

¹⁰³<https://www.mozilla.org/uk/firefox/new/>

¹⁰⁴<http://getfirebug.com/>



Сторінка Firebug

Основа HTML документа

В цій секції ми з вами підготуємо базу HTML документа, наб'ємо демо контент та підключимо фреймворк Twitter Bootstrap до нашої сторінки.

Заготовка

Для початку створимо папочку у файловій системі, куди покладемо наш порожній index.html файлік. Він міститиме весь HTML код необхідний для сторінки списку студентів.

Я впевнений, що ви знаєте як створити папку та порожній файл у ній в себе на комп’ютері віконними засобами, тому демонструю усі операції з файловою системою виключно в командній стрічці:

```
1 # я триматиму папочку у себе на робочому столі;
2 # ~ в лінуксі і макінтоші позначає домашню папку користувача:
3 $ cd ~/Desktop
4
5 # створюю папку static і заходжу у неї
6 $ mkdir static
7 $ cd static
8
9 # створюю порожній файл index.html
10 touch index.html
```

Відкривайте його у своєму редакторі коду та починайте набивати базою - обов'язковими елементами, які присутні у будь-якому HTML елементі.

В HTML мові коментарі (елементи для документації, що ігноруються інтерпретатором мови) позначаються як “`<!-- коментар -->`”.

База HTML документа

```
1 <!DOCTYPE html>
2 <html lang="uk">
3   <head>
4     </head>
5   <body>
6     </body>
7 </html>
```

Проаналізуємо даний код рядочок за рядочком:

- 1ий: декларація типу документу; ми використали тип `html`, який означає, що ми використовуватимемо HTML5 стандарт у нашему документі; більше про типи документів [тут¹⁰⁵](#);

¹⁰⁵<http://htmlbook.ru/html/doctype>

- 2ий: html - вмістилище усіх тегів на сторінці; ми додали атрибут lang із значенням uk, який повідомляє браузер про те, що документ містить текст українською мовою;
- 3ий: head - хідер документа; містить теги, які є поза видимою частиною сторінки в браузері; head містить мета інформацію про сторінку та включення статичних ресурсів (css, javascript);
- 5ий: body - тіло документа; містить видиму частину сторінки в браузері; це усі теги, що містять певний контент для користувача;
- я роблю по 2 пробіли відступів між кожним вкладеним тегом і його батьківським тегом; це покращує читабельність коду; відступи не є обов'язковим елементом мови HTML;
- усі теги в даному прикладі, крім DOCTYPE декларації, мають відповідний закриваючий тег.

Додамо ще кілька видимих тегів до нашої сторінки, а також кілька мета тегів про нашу сторінку:

Демо контент та базові мета теги

```
1 <!DOCTYPE html>
2 <html lang="uk">
3
4   <head>
5     <meta charset="UTF-8"/>
6     <title>Сервіс Обліку Студентів</title>
7     <meta name="description"
8       value="Система Обліку Студентів Навчального Закладу" />
9   </head>
10
11   <body>
12     <h1>Вітаю вас на нашій сторінці!</h1>
13     <p>Поки тут порожньо, але незабаром ми матимемо список студентів\>
14   </p>
15   </body>
16
17 </html>
```

Знову пройдемось по нових тегах з прив'язкою до рядочків:

- 5ий: meta - вказує веб-переглядачу у якому кодуванні текст даного документу; якщо даного тегу не вказати, тоді можемо бачити “іерогліфи” замість кирилиці; важливо мати цей тег перший на сторінці в тезі head;
- 6ий: title - вказує мета заголовок сторінки; використовується пошуковими сервісами та браузером для відображення у закладці;
- 7ий: meta - ще один мета тег; його назва (атрибут name) вказує на description, що означає, що даний тег встановлює мета опис даної сторінки; важливий для SEO;
- 12ий: h1 - тег заголовка, видимий на сторінці як більший текст; це заголовок першого рівня, всього є 6 рівнів заголовків h1-h6; на сторінці може бути лише один h1 тег і багато заголовків нижчих рівнів;
- 13ий: p - параграф тексту.

Запам'ятайте. Ваш текстовий редактор в 99% має зберігати текстові файли в UTF-8 кодуванні. Будь-то Python, HTML, CSS чи інша мова.



Демо HTML сторінка як вона виглядає у браузері

Підключаємо Twitter Bootstrap

Тепер підключимо на сторінку фреймворк Twitter Bootstrap.

Зайшовши на його сторінку [швидкого старту](#)¹⁰⁶ можемо ознайомитись із кількома можливими способами підключення даної бібліотеки на нашу сторінку.

¹⁰⁶ <http://getbootstrap.com/getting-started/>

По старинці ми б мали завантажити архів із CSS та Javascript файлами, розпакувати його у потрібну нам папочку і тоді з нашого HTML документу залінкувати необхідні файлики.

Але тепер є популярними [CDN¹⁰⁷](#) ресурси (Мережі Доставки Контенту). Це мережі, що дають нам швидкий доступ до статичних ресурсів (css, javascript, зображення) таким чином пришвидшуячи наш вебсайт. Практично кожна популярна бібліотека знаходиться на одному із доступних публічних CDN-ів. Тому не потрібно їх ставити локально, а просто підключити їх ззовні.

Для Twitter Bootstrap нам треба jQuery бібліотеку, Twitter Bootstrap Javascript файл, а також один CSS файл. Лінки на усі ресурси я просто шукаю з допомогою Google і вибираю найпопулярніші CDN-и.

Ось вигляд нашого HTML документу з уже під'єднаними CSS та Javascript ресурсами для правильної роботи Twitter Bootstrap:

Підключений Twitter Bootstrap

```
1 <!DOCTYPE html>
2 <html lang="uk">
3
4     <head>
5         <meta charset="UTF-8" />
6         <title>Сервіс Обліку Студентів</title>
7         <meta name="description"
8             value="Система Обліку Студентів Навчального Закладу" />
9
10        <!-- Include Styles -->
11        <link rel="stylesheet"
12            href="https://cdn.jsdelivr.net/bootstrap/3.3.0/css/bootstrap\.
13 min.css">
14
15    </head>
16
17    <body>
18        <h1>Вітаю вас на нашій сторінці!</h1>
```

¹⁰⁷http://ru.wikipedia.org/wiki/Content_Delivery_Network

```
19      <p>Поки тут порожньо, але незабаром ми матимемо список студентів\</p>
20
21
22      <!-- Javascripts Section -->
23      <script src="http://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.1/\>
24 jquery.js"></script>
25      <script src="https://cdn.jsdelivr.net/bootstrap/3.3.0/js/bootstrap\>
26 ap.min.js"></script>
27
28  </body>
29
30 </html>
```

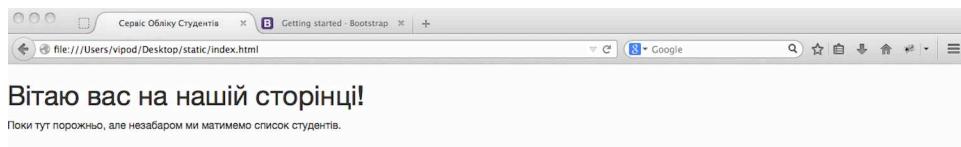
Що ж змінилось у нашому HTML документі:

- 10ий: Include Styles - коментарі завжди вітаються; я намагаюсь коментувати HTML також, адже в реальних проектах шаблони зазвичай мають великі розміри і тоді без правильних відступів та коментарів про секції в документі, можна загубитись;
- 11ий: link - стилі в даному випадку включаємо з допомогою тегу link та атрибути rel="stylesheet" (relation - зв'язок), який вказує браузеру, що це лінк на таблицю стилів; атрибут href вказує на адресу, під якою живуть стилі; ми взяли стилі Twitter Bootstrap із зовнішнього ресурсу - CDN; стилі ми підключаємо в тегу head зазвичай;
- 22ий: Javascripts Section - знову коментар про початок включення Javascript файлів на сторінці;
- 23ій: script - підключаємо перший Javascript файл з допомогою тегу script; src вказує на зовнішній файл jquery.js, без якого Twitter Bootstrap Javascript код не зможе працювати;
- 25ий: script - другий файл підключає вже Javascript код самої бібліотеки Twitter Bootstrap.

Якщо бачите, що лінки на деякі з Javascript чи CSS ресурсів не працюють, спочатку перевірте чи правильно ви викопіювали їх із книги. Інструмент верстки книги автоматично додає символ зворотнього слеша до тексту, що не влазиться в один рядок. Тому, взагалі, рекомендую копіювати код не з книги, а з коду, що йде з книгою.

Javascript файли, згідно останніх практик, рекомендується підключати вкінці сторінки одразу перед закриваючим тегом body. Це потрібно, щоб браузер швидше відображав сторінку і завантаження Javascript файлів та їх запуск не блокував збірку інших ресурсів на сторінці.

Після підключення Twitter Bootstrap можете перезавантажити сторінку у браузері і одразу побачити різницю в стилях параграфа тексту та заголовка:



Демо сторінка HTML з Twitter Bootstrap

Перед переходом до наступної секції, де ми зверстаємо верхню частину сторінки нашого проекту, пропоную самостійно ознайомитись із набором функціоналу Twitter Bootstrap.

Тепер ми готові до реальної роботи!

Шапка та футер

Огляд

В цій секції ми в першу чергу розберемось із Grid системою Twitter Bootstrap фреймворка. З її допомогою ми зможемо з легкістю розташовувати елементи на сторінці у потрібних нам місцях.

А також зверстаємо верхню частину (шапку) нашої головної сторінки. Ось які елементи входять у нашу шапку та нижній колонитул:

The screenshot shows a web application for managing students. At the top, there's a header bar with the title "Сервіс Обліку Студентів". Below the title are three navigation tabs: "Студенти" (highlighted with a red arrow), "Відвідування" (highlighted with a red arrow), and "Групи". To the right of the tabs is a dropdown menu labeled "Група: Усі Студенти". The main content area contains a message "Студент успішно доданий." Below this is a table titled "База Студентів" with columns: "#", "Фото", "Прізвище ↑", "Ім'я", "№ Білету", and "Дії". Three student records are listed: 1. Корост Андрій 2123; 2. Подоба Віталій 254; 3. Притула Тарас 5332. On the right side of the table, there's a "Дії" dropdown menu with options: "Редагувати", "Відвідування", and "Видалити". At the bottom of the page is a footer with the text "© 2014 Сервіс Обліку Studentів". Red arrows point from the text labels below to specific parts of the interface: "НАВІГАЦІЯ" points to the tabs, "ЗАГОЛОВОК - ЛОГО" points to the header title, and "СЕЛЕКТОР СТУДЕНТИВ" points to the group dropdown.

Елементи шапки та футера сторінки

Таким чином маємо:

- текстове лого;
- випадайка (селектор) із групами;
- головне навігаційне меню аплікації;
- нижній колонитул (футер) із текстом.

Grid Система

Grid з англ. - це сітка. CSS фреймворки полегшують розробникам розташовувати елементи на сторінці (іншими словами визначати лейаут (з англ. layout - планування)) за допомогою поділу сторінки на уявну сітку. Тоді нам, як розробникам, залишається лише вказати в якій комірці даної сітки вставити той чи інший елемент сторінки. Наприклад Twitter Bootstrap сітка ділить сторінку на 12 вертикальних колонок. Кількість рядків ми визначаємо довільно.

Поділимо нашу сторінку умовно на рядочки:

The screenshot shows a web application titled "Сервіс Обліку Студентів". At the top right, there is a dropdown menu set to "Усі Студенти". Below the title, there are three navigation links: "Студенти", "Відвідування", and "Групи". A success message "Студент успішно доданий." is displayed in a yellow bar. The main content area is titled "База Студентів" and contains a table with student data. The table has columns: "#", "Фото", "Прізвище", "Ім'я", "№ Білету", and "Дії". Three students are listed:

#	Фото	Прізвище	Ім'я	№ Білету	Дії
1		Корост	Андрій	2123	<button>Дія ▾</button>
2		Подоба	Віталій	254	<button>Дія ▾</button>
3		Притула	Тарас	5332	<button>Дія ▾</button>

At the bottom left, there is a copyright notice: "© 2014 Сервіс Обліку Студентів". On the right side, there is a small floating menu with options: "Редагувати", "Відвідування", and "Видалити".

Рядки головної сторінки

Наша сторінка доволі проста як щодо кількості елементів, так і щодо їх розташування. По-суті виходить 4 рядочка і лише один із них має більше, ніж одну колонку - це перший рядок з текстом-логотипом та меню груп.

Усі елементи безпосередньо списку студентів ми побудуємо з допомогою HTML таблиці, відповідно там нам Grid система не пригодиться.

На домашнє завдання можете перенести кнопку “Додати студента” в один рядочок із заголовком “База Студентів”. Так буде наша сторінка більш компактною.

Гляньте на [Grid систему¹⁰⁸](#) Twitter Bootstrap. Є кілька важливих правил як писати свій HTML код сторінки, щоб скористатись їхньою сіткою:

- рядки мають бути всередині елементу з класом “container”;
- в “container” можуть бути лише рядки; рядок - це елемент із класом “row”;
- в рядках можуть бути лише колонки; колонка - це елемент із класом “col-xs-4”;
- xs (extra small) в назві класу колонки може також бути sm (small), md (medium), lg (large), взалежності від розміру екрану, для яких ми пишемо нашу верстку;
- число в класі колонки означає скільки клітинок в рядочку займатиме дана колонка; Grid система в Twitter Bootstrap ділить сторінку на 12 рівних колонок; тому колонка може займати максимум 12 клітинок сітки, а сума усіх клітинок, що займають кілька колонок в одному рядку має бути рівна 12; напр. якщо ми хочемо мати в одному рядку два елементи, один з яких займає 2/3 сторінки, а інший, відповідно, 1/3, тоді клас у першого має бути “col-xs-8”, а у другого - “col-xs-4”.

Давайте поділимо нашу попередньо заготовану демо сторінку на Twitter Bootstrap сітку згідно вище наведеного зображення:

Сітка для головної сторінки

```

1 <!DOCTYPE html>
2 <html lang="uk">
3
4   <head>
5     <meta charset="UTF-8"/>
6     <title>Сервіс Обліку Студентів</title>
7     <meta name="description"
8       value="Система Обліку Студентів Навчального Закладу" />
9
10    <!-- Include Styles -->
11    <link rel="stylesheet"
```

¹⁰⁸<http://getbootstrap.com/css/#grid>

```
12         href="https://cdn.jsdelivr.net/bootstrap/3.3.0/css/bootstrap\\
13 ap.min.css">
14
15     <style type="text/css">.col-xs-12, .col-xs-4, .col-xs-8 {border:\\
16 1px solid red;}</style>
17
18     </head>
19
20     <body>
21
22         <!-- Start Container -->
23         <div class="container">
24
25             <!-- Start Header -->
26             <div class="row" id="header">
27
28                 <!-- Logo -->
29                 <div class="col-xs-8">
30                     Лого
31                 </div>
32
33                 <!-- Groups Selector -->
34                 <div class="col-xs-4" id="group-selector">
35                     Селектор Груп
36                 </div>
37             </div>
38             <!-- End Header -->
39
40             <!-- Start SubHeader -->
41             <div class="row" id="sub-header">
42                 <div class="col-xs-12">
43                     Навігація
44                 </div>
45             </div>
46             <!-- End SubHeader -->
```

```
47
48     <!-- Start Main Page Content -->
49     <div class="row" id="content-columns">
50
51         <div class="col-xs-12" id="content-column">
52             Контент, Список Студентів
53         </div>
54
55     </div>
56     <!-- End Main Page Content -->
57
58     <!-- Start Footer -->
59     <div class="row" id="footer">
60         <div class="col-xs-12">
61             Footer
62         </div>
63     </div>
64     <!-- End Footer -->
65
66     </div>
67     <!-- End Container -->
68
69
70     <!-- Javascripts Section -->
71     <script src="http://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.1/\
72 jquery.js"></script> \
73
74     <script src="https://cdn.jsdelivr.net/bootstrap/3.3.0/js/bootstrap\
75 min.js"></script>
76
77     </body>
78
79 </html>
```

Проаналізуємо основні моменти оновленого HTML коду нашого index.html

файла по рядках:

- 15ий: style - тимчасові стилі для того, щоб показати межі клітинок червоним кольором; зауважте я скористався тегом style для вкладення на сторінку коду стилів напряму, а не із зовнішнього файлу, як ми це робили з допомогою тегу link;
- 23ій: div.container: усі видимі елементи нашої сторінки будуть тепер всередині даного контейнера; всередині нього можуть бути лише рядочки;
- 26ий: div.row#header: перший рядок, що є в div.container; у нього ми пізніше вкладемо текстове лого та меню груп; йому ми також додали унікальний атрибут id, щоб пізніше можна було його однозначно ідентифікувати на сторінці; class не є унікальним ідентифікатором, а зазвичай служить для стилів елементу, в той час як id може бути лише один на сторінці; class дорівнює row, що означає, що даний елемент позначає рядок в нашему лейауті;
- 29ий: div.col-xs-8: перша колонка в рядку div#header; займає перших 8 клітинок із 12, тобто 2/3 ширини div.container; в нього ми пізніше вкладемо наше текстове лого;
- 34ий: div.cols-xs-4#group-selector: друга колонка в нашему першому рядочку; займає решту ширини рядка, тобто 4 клітинки (всього 12 клітинок є у кожному рядку); ми додали також id, щоб пізніше “навішати” Javascript функціонал під час вибору групи;
- 51ий: div.col-xs-12#content-column: головна колонка, що міститиме список студентів; займає усю ширину контейнера.

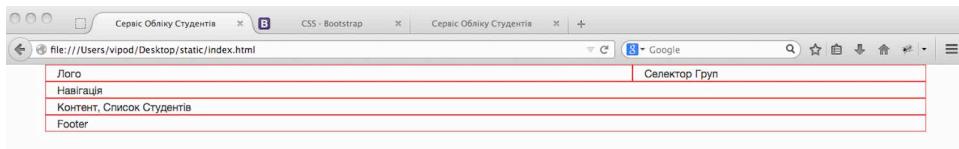
Зверніть увагу, що я використав для нашої сітки теги div. Це блочний елемент призначений для групування елементів. Найкраще підходить для побудови лейаута на сторінці.

Блочні елементи (block) - це елементи, які займають увесь рядок на сторінці. Тобто займають усю ширину батьківського елементу. Вони зазвичай починаються з нового рядка. Блочними є h1-h6, div, form, table, ul/ol, і т.д.

Рядкові елементи (inline) - за замовчуванням йдуть в тексті підряд, без розриву і перенесення рядка. Ширина рядкового елемента дорівнює ширині його вмісту. До рядкових елементів належать a, span, button, input, img, і т.д.

Зауважте, я намагаюсь побільше додавати коментарів для позначок окремих секцій сторінки. З ростом коду вони нам дуже допоможуть орієнтуватись у великому HTML файлі.

Результатом буде:



Тест лейауту головної сторінки

Як бачите з допомогою сітки від Twitter Bootstrap без жодного власного рядочку CSS коду ми змогли швиденько розташувати наші елементи на потрібні місця на сторінці.

Залишилось заповнити наш лейаут необхідними елементами. Тепер можемо забрати наші тимчасові стилі із червоними межами для комірок і рухатись до наступної секції.

Заголовок і навігація

Із текстовим лого все доволі просто. Зробимо його заголовком першого рівня:

```

28      <!-- Logo -->
29      <div class="col-xs-8">
30          <h1>Сервіс Обліку Студентів</h1>
31      </div>

```

А от для навігації ми скористаємось готовим [навігаційним віджетом¹⁰⁹](#) із Twitter Bootstrap. Просто викопіюємо кусок HTML коду звідти і змінимо текст всередині лінків (табів):

```

28      <!-- Start SubHeader -->
29      <div class="row" id="sub-header">
30          <div class="col-xs-12">
31              <ul class="nav nav-tabs" role="tablist">
32                  <li role="presentation" class="active"><a href="#">Студенти</a></li>
33                  <li role="presentation"><a href="#">Відвідування</a></li>
34                  <li role="presentation"><a href="#">Групи</a></li>
35          </ul>
36      </div>
37  </div>
38      <!-- End SubHeader -->

```

У прикладі вище ми вставили тег ul (unordered list, список без нумерації) із спеціальними класами та атрибутом role. Вклали у нього елементи списку li (list item, елемент списку) з атрибутом role рівним стрічці presentation. Вам не потрібно достеменно знати як це працює. Просто скажу, що Javascript та CSS код фреймворку Twitter Bootstrap спеціальним чином обробляє дані функціональні атрибути та застосовує певну поведінку, відповідно до типу віджета.

Досить просто. Всього лише застосували потрібні атрибути, класи до html тегів і маємо чудовий результат без додаткового CSS чи Javascript коду. Це і є суть Twitter Bootstrap фреймворку.

¹⁰⁹<http://getbootstrap.com/components/#nav>

Меню груп

Для меню груп просто скористаємо полем форми під назвою select. Він дасть нам випадаюче меню з набором опцій:

Меню груп

```
33      <!-- Groups Selector -->
34      <div class="col-xs-4" id="group-selector">
35          <strong>Група:</strong>
36          <select>
37              <option value="">Усі Студенти</option>
38              <option value="">МтМ - 21, Подоба Віталій (№ 254)</option>
39      n>
40              <option value="">МтМ - 22, Корост Андрій (№ 2123)</option>
41      n>
42          </select>
43      </div>
44  </div>
45  <!-- End Header -->
```

Як бачите, меню групи складається з двох частин: мітки “Група” та самого меню. Текст “Група” ми огорнули в тег strong, щоб текст був жирним шрифтом.

Потім йде select, який містить список опцій. Кожна опція має атрибут value, значення якого приходить на сервер у випадку, якщо опція була обрана. Крім того тег option містить текст всередині. Він призначений в першу чергу для користувачького інтерфейсу. Проте, якщо немає атрибути value, тоді вміст тегу option прийде на сервер замість нього.

Футер

В даній секції нам залишилось оновити нижній колонтитул. В принципі там уже все готове. Потрібно лише оновити текст:

Футер оновлено

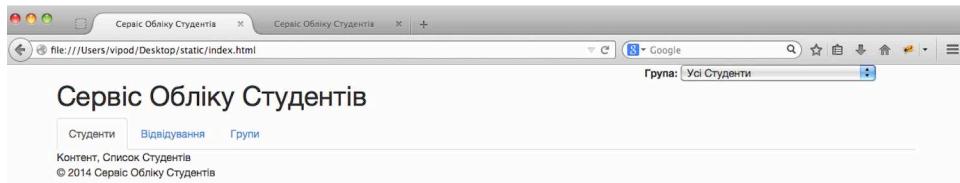
```

58      <!-- Start Footer -->
59      <div class="row" id="footer">
60          <div class="col-xs-12">
61              &copy; 2014 Сервіс Обліку Студентів
62          </div>
63      </div>
64      <!-- End Footer -->

```

Єдине на що хочу звернути вашу увагу у коді вище це те, що маємо особливий елемент “©”. Це називається [HTML Entity¹¹⁰](#) і він вставляє значок копірайтингу у наш футер.

Ось як виглядає сторінка на даний момент:



Шапка майже готова

Ще залишились деякі деталі, якщо порівнювати із оригінальними специфікаціями проекту, але це ми залишимо на закінчення глави. Або, як варіант, можете спробувати доробити їх ще до того, як ми до них доберемось в книзі. Дані правки вимагають кастомного CSS коду.

Список студентів

Добралися до найважливішої частини нашої сторінки - списку студентів.

Наступні елементи увійдуть у нашу найбільшу комірку у сітці:

- статусне повідомлення: “Зміни скасовано”;

¹¹⁰<http://bit.ly/vphmentities>

- заголовок сторінки: “База Студентів”;
- кнопка “Додати Студента”;
- таблиця із списком студентів;
- посторінкова навігація по списку студентів.

Статусне повідомлення, заголовок і кнопка додавання студента

Усі ці елементи доволі прості, тому зробимо їх одним махом у даній секції.

Почнемо із статусного повідомлення. Це елемент, який буде показуватись користувачу після виконання певних дій, як от додавання студента чи скасування операції.

Для цього скористаємось заготовочкою із Twitter Bootstrap під назвою [Alerts¹¹¹](#) (нотифікації, повідомлення). Використаємо той, що із жовтеньким кольором, щоб краще підходив під наш визначений дизайн. У нього клас “alert-warning”:

Додаємо статусне повідомлення

```

1      <!-- Start Main Page Content -->
2      <div class="row" id="content-columns">
3
4          <div class="col-xs-12" id="content-column">
5
6              <!-- Status Message -->
7              <div class="alert alert-warning">Зміни скасовано.</div>
8
9          </div>
10
11      </div>
12      <!-- End Main Page Content -->

```

На 7-му рядочку ми додаємо новий елемент починаючи із коментаря звісно. Так, просто вставивши div із класом “alert” та “alert-warning” ми отримаємо текст огорнутий в гарну рамочку і жовтий фон.

¹¹¹<http://getbootstrap.com/components/#alerts>

Далі маємо заголовок сторінки. Оскільки ми уже маємо заголовок першого рівня в лого, то наш заголовок сторінки може бути лише другого рівня:

Додаємо заголовок сторінки

```
48      <!-- Start Main Page Content -->
49      <div class="row" id="content-columns">
50
51          <div class="col-xs-12" id="content-column">
52
53              <!-- Status Message -->
54              <div class="alert alert-warning">Зміни скасовано.</div>
55
56              <h2>База Студентів</h2>
57
58      </div>
59
60  </div>
61  <!-- End Main Page Content -->
```

На 56-му рядочку ми додали h2 тег із текстом База Студентів.

Останній елемент, що нам треба додати до головної колонки на сторінці перед тим як перейти до таблиці студентів - кнопка додавання студента. На вигляд це кнопка, але насправді тут достатньо лінка, адже ми маємо просто перейти на форму додавання нового студента при кліку по ній. А для такої дії достатньо запиту типу GET.

Коли ми навігуємо у вебі, клікаємо лінки, завантажуємо сайти набрані в URL адресі нашого браузера, тоді ми постійно виконуємо запити на сервер типу GET. Тобто отримуємо інформацію із сервера. А коли ми працюємо із формами, зокрема відправляємо дані з форми на сервер, зазвичай тип запиту є POST. Тобто ми відправляємо дані на сервер, а не лише отримуємо дані із сервера. Кнопки на веб сторінках зазвичай використовуються у

формах і переважно для POST запитів.

Тому дану кнопку реалізуємо з допомогою тегу лінка “a”, а вигляд кнопки нам зробить [Twitter Bootstrap](#)¹¹²:

Додаємо кнопку

```
48      <!-- Start Main Page Content -->
49      <div class="row" id="content-columns">
50
51          <div class="col-xs-12" id="content-column">
52
53              <!-- Status Message -->
54              <div class="alert alert-warning">Зміни скасовано.</div>
55
56              <h2>База Студентів</h2>
57
58              <!-- Add Student Button -->
59              <a class="btn btn-primary" href="#">Додати Студента</a>
60
61              <div>Тут буде таблиця студентів</div>
62
63          </div>
64
65      </div>
66      <!-- End Main Page Content -->
```

В 59-му рядку можете бачити новий тег “a” із класами “btn” та “btn-primary”. Ці класи дають нам лінк у вигляді синьої кнопки. Майже як у початковому дизайні. На даний момент атрибут href вказує на решітку (#) - недійсне посилання, яке ми виправимо у наступних главах.

Також я додав місце, де буде таблиця із студентами.

¹¹²<http://getbootstrap.com/css/#buttons>

Ще залишились кілька дрібних нюансів, щоб досягнути вигляду як в дизайні, але ми їх залишимо на завершення даної глави. А в даний момент ось як виглядає контент нашої сторінки:

Таблиця студентів

Переходимо до найцікавішого. Список студентів зробимо через таблицю, адже його вигляд ідеально підходить у табличні дані із рядками та колонками.

Почнемо із простого: наб'ємо початкову структуру таблиці:

Структура таблиці для студентів

```

61      <!-- Start Students Listing -->
62      <table>
63          <thead>
64              <tr>
65                  <th>№</th>
66                  <th>Фото</th>
67                  <th>Прізвище</th>
68                  <th>Ім'я</th>
69                  <th>№ Білету</th>
70                  <th>Дії</th>
71          </tr>
72      </thead>
73      <tbody>
```

```

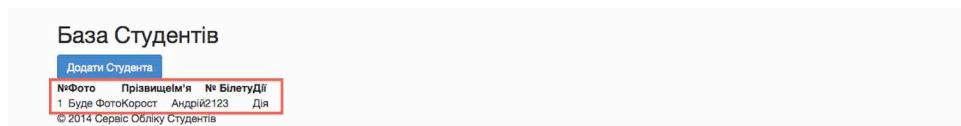
74 <tr>
75   <td>1</td>
76   <td>Буде Фото</td>
77   <td>Корост</td>
78   <td>Андрій</td>
79   <td>2123</td>
80   <td>Дія</td>
81 </tr>
82 </tbody>
83 </table>
84 <!-- End Students Listing -->

```

Хочу підкреслити кілька моментів по деяких рядках:

- 62ий: table - вперше скористаємося тегом таблиці; цей тег має масу атрибутів та властивостей, але вони нас поки не цікавлять, адже увесь наступний вигляд ми “магічно” застосуємо кількома класами з Twitter Bootstrap;
- 63ій: thead - містилище заголовкових рядочків таблиці, так званий хідер таблиці;
- 64ий: tr - рядок таблиці, одинаковий як для тіла таблиці, так і для хідера таблиці;
- 65ий: th - тег однієї комірки таблиці саме в шапці таблиці; по замовчуванню він застосовує жирний шрифт до внутрішнього тексту;
- 73ій: tbody - тіло таблиці, містить рядочки тіла таблиці;
- 75ий: td - тег однієї комірки таблиці, але вже тіла таблиці; не застосовує додаткових стилів до внутрішніх елементів.

На даний момент таблиця виглядає жахливо у нашому веб-браузері:



The screenshot shows a web page titled "База Студентів". At the top, there is a blue button labeled "Додати Студента". Below it is a table with one row, outlined in red. The table has four columns: "№Фото", "Прізвищем'я", "№ Білету", and "Дія". The first column contains a small thumbnail image. The second column contains the name "Андрій". The third column contains the number "2123". The fourth column contains the word "Дія". At the bottom of the table, there is a small note: "© 2014 Сервіс Обліку Студентів".

Дефолтний вигляд таблиці

Але кількома простими рухами та з допомогою [стилів таблиць¹¹³](#) із Twitter Bootstrap:

Додаємо класи до таблиці

```
61      <!-- Start Students Listing -->
62      <table class="table table-hover table-striped">
```

наша таблиця перетворюється у справжній витвір мистецтва :-):

База Студентів

База Студентів					
Додати Студента					
№	Фото	Прізвище	Ім'я	№ Білету	Дії
1	Буде Фото	Корост	Андрій	2123	Дія

© 2014 Сервіс Обліку Студентів

Twitter Bootstrap і Таблиця

А все, що ми зробили це додали класи:

- .table - який ддав нам гарні відступи всередині комірок та горизонтальні розділювачі між рядками;
- .table-hover - гарний ефект зміни фону при наведенні мишкою над рядками таблиці;
- .table-striped - чергування фонів рядків таблиці, так званий ефект зебри для кращого сприйняття таблиці візуально.

Що залишилось? Потрібно текст в кількох комірках зробити лінками, додати зображення, а також зробити меню дій навпроти кожного студента.

Спочатку потрібно підготувати кілька зображень, які ми використаємо в якості зображень студентів. Як завжди я демонструю варіант з командною стрічкою, ви ж можете користуватися також графічним інтерфейсом вашої операційної системи для управління файлами і папками:

¹¹³<http://getbootstrap.com/css/#tables>

Готуємо зображення

```
1 # заходимо у нашу папку static
2 $ cd ~/Desktop/static
3
4 # створюємо папку img для наших зображень
5 $ mkdir img
6
7 # копіюємо сюди 3 будь-які зображення, я вибрав свої власні
8 # зображення наперед заготовані з іншої папочки
9 $ cp ~/Pictures/me/me.jpeg img/
10 $ cp ~/Pictures/me/podoba3.jpg img/
11 $ cp ~/Pictures/me/piv.png img/
12
13 # перевіримо чи дійсно вони скопіювались
14 $ ls img/
15 me.jpeg     piv.png     podoba3.jpg
```

Тепер ми можемо засилатись на дані зображення із нашого HTML документа наступним чином: “./img/me.jpeg”. Це локальне посилання на файл з файлової системи. Використавши тер img та маючи заготовані зображення, ми зможемо вставити наші зображення у потрібному місці.

Одна крапка (.) означає поточну директорію. Дві крапки (..) означають директорію на рівень вище.

Для випадайки із списком дій над студентом ми скористаємось черговим класним [віджетом](#)¹¹⁴ із Twitter Bootstrap. Просто скопіюємо один до одного код-заготовку цього віджета випадайки та змінимо елементи в меню на ті, що нам потрібні.

Ну тепер ми уже підготувалися, щоб завершити верстку елементів всередині нашої таблиці:

¹¹⁴<http://getbootstrap.com/components/#dropdowns>

Таблиця практично готова

```
61      <!-- Start Students Listing -->
62      <table class="table table-hover table-striped">
63          <thead>
64              <tr>
65                  <th><a href="">№</a></th>
66                  <th>Фото</th>
67                  <th><a href="">Прізвище &uarr;</a></th>
68                  <th><a href="#">Ім'я</a></th>
69                  <th><a href="#">№ Білету</a></th>
70                  <th>Дії</th>
71          </thead>
72          <tbody>
73              <tr>
74                  <td>1</td>
75                  <td>\ 
76          </td>
77                  <td><a title="Редагувати" href="#">Корост</a></td>
78                  <td><a title="Редагувати" href="#">Андрій</a></td>
79                  <td>2123</td>
80                  <td>
81                      <div class="dropdown">
82                          <button class="btn btn-default dropdown-toggle"
83                              type="button" data-toggle="dropdown">Дія
84                          <span class="caret"></span>
85                      </button>
86                      <ul class="dropdown-menu" role="menu">
87                          <li role="presentation">
88                              <a role="menuitem" tabindex="-1"
89                                  href="#">Відвідування</a>
90                          </li>
91                          <li role="presentation">
92                              <a role="menuitem" tabindex="-1" href="#">Pe\
93          дагувати</a>
94                      </li>
```

```
95          <li role="presentation">
96              <a role="menuitem" tabindex="-1" href="#">Ви\
97      далити</a>
98          </li>
99      </ul>
100     </div>
101     </td>
102     </tr>
103     </tbody>
104     </table>
105     <!-- End Students Listing -->
```

Давайте по рядках розглянемо оновлені елементи таблиці:

- 67ий: a - тут ми огорнули текст “Прізвище” у тег a; далі в наступних главах ми зробимо так, щоб клік по заголовках №, Прізвище, Ім’я та № Білету будуть сортувати студентів по відповідних полях;
- 75ий: img - ми вставили тег зображення, обмежили висоту і ширину до 30 пікселів та вказали (атрибут src) шлях до потрібного зображення;
- 77ий: a - Прізвище та Ім’я студентів ми огорнули також в лінки (тег a); у наступних главах ми зробимо так, щоб при кліку по даних посиланнях користувач переходив на форму редагування студента;
- 81ий: div.dropdown - цей весь кусок коду ми викопіювали із Twitter Bootstrap прикладу і змінили список елементів в меню; для кнопки Дія використовується тег button, а для самого меню невпорядкований список ul; з допомогою спеціальних класів та атрибутів Twitter Bootstrap стилі та Javascript код прив’яжуть увесь необхідний функціонал до нашої випадайки; а лінки в меню ми пізніше оновимо, щоб вказували на потрібні сторінки нашої аплікації.

В таблиці у нас, як бачите, ще поки зображення не круглі як у дизайні. Але це ми залишимо на пізніше:

База Студентів					
Додати Студента					
№	Фото	Прізвище ↑	Ім'я	№ Білету	Дії
1		Корост	Андрій	2123	Дія ▾

© 2014 Сервіс Обліку Студентів

Майже готова таблиця студентів

На самостійне опрацювання: додати ще два рядочки в таблицю із студентами згідно із дизайном.

Посторінкова навігація

Перед тим як переходити до дрібних фіксів, додамо ще одну річ, якої немає в початковому дизайні - посторінкову навігію для списку студентів.

Її ми також знайдемо серед [заготовок](#)¹¹⁵ у Twitter Bootstrap. Додамо її одразу після закриваючого тегу </table>:

Посторінкова Навігація

```

107   <nav>
108     <ul class="pagination">
109       <li><a href="#">&laquo;</a></li>
110       <li><a href="#">1</a></li>
111       <li class="active"><a href="#">2</a></li>
112       <li><a href="#">3</a></li>
113       <li><a href="#">4</a></li>
114       <li><a href="#">5</a></li>
115       <li><a href="#">&raquo;</a></li>
116     </ul>
117   </nav>

```

¹¹⁵<http://getbootstrap.com/components/#pagination>

Єдине, що я зробив у викопіюваному куску коду для посторінкової навігації - це додав клас “active” до 2ї сторінки навігації, щоб мати приклад вигляду обраної сторінки.

Цей віджет цікавий тим, що він використовує HTML5 тег - nav. Це семантичний елемент, який вказує браузеру, що він містить навігаційний елемент всередині.

Сама навігація доволі проста. Це знову ж таки невпорядкований список елементів ul. В кожному із елементів списку маємо посилання (тег a). З обидвох сторін навігації маємо подвійні стрілочки вліво та вправо. Для них ми використали HTML Entity.

№	Фото	Прізвище	Ім'я	№ Білету	Дії
1		Корошт	Андрій	2123	<button>Дія</button>
2		Подоба	Віталій	254	<button>Дія</button>
3		Притула	Тарас	5332	<button>Дія</button>

Сторінка Списку Студентів Майже Готова!

Залишки

В цій секції ми вперше з вами напишемо (якщо не рахувати тих тимчасових стилей для червоного кольору меж комірок для тесту сітки) кілька рядочків CSS коду.

Маємо кілька невеликих нюансів, що віддаляють нас від кінцевого вигляду, який вказаний у початковому дизайні:

- заголовки в дизайні є жирним шрифтом;
- наше меню груп потребує трохи позиціонування та оновлення для крашого вигляду;
- відстань навколо статусного повідомлення виглядає не дуже;
- зображення в таблиці не є круглими;
- бракує верхньої лінії над футером.

Першим ділом створимо наш власний файлик main.css в підпапці css, куди будемо записувати наші кастомні стилі:

Створюємо main.css

```
1 $ cd ~/Desktop/static  
2 $ mkdir css  
3 $ touch css/main.css
```

Тепер під'єднаємо його до нашого HTML документу подібно до того, як ми це зробили із CSS файлом з Twitter Bootstrap фреймворку:

Підключаємо main.css

```
4 <head>  
5   <meta charset="UTF-8" />  
6   <title>Сервіс Обліку Студентів</title>  
7   <meta name="description"  
8     value="Система Обліку Студентів Навчального Закладу" />  
9  
10  <!-- Include Styles -->  
11  <link rel="stylesheet"  
12    href="https://cdn.jsdelivr.net/npm/bootstrap@3.3.0/css/bootstrap.min.css" />  
13 <link rel="stylesheet" href="./css/main.css" />  
14  
15 </head>
```

Зверніть увагу, що я вклав новий link тег після існуючого тегу. Це зроблено для того, щоб наші кастомні стилі мали останнє слово у визначені стилів для елементів на сторінці. Тобто порядок включення стилів (так само і Javascript коду) має значення!

Тепер відкривайте main.css файл у вашому улюбленаому редакторі і будемо його по-трохи оновлювати нашими правилами. А почнемо ми із заголовків. Зробимо їх усіх жирним шрифтом. Заголовки є до шести рівнів: h1-h6.

Жирний шрифт для усіх заголовків

```
1 /* Global Styles */  
2 h1, h2, h3, h4, h5, h6 {  
3   font-weight: bold;  
4 }
```

Як бачите, я також додаю коментарі до CSS коду. Згодом вони допоможуть легше орієнтуватись у великому CSS файлі. Рекомендую вам також притримуватись цього правила.

Для одного набору правил може бути більше, ніж один селектор. Усі вони мають бути розділені комами. Головне не забувайте, що перед фігурною дужкою селектор не може мати коми. Приклад: "h1, h2, h3, h4 {".

У вище наведеному правилі ми дали властивості font-weight (вага шрифта) значення bold (жирний, грубий). Тепер усі заголовки на нашій сторінці будуть жирним шрифтом.

Далі ми посунемо наше статусне повідомлення, щоб воно так не “приліпало” до навігаційної панельки. Для цього додамо зовнішню відстань (margin) у кілька пікселів:

Додаємо зовнішню відстань до статусного повідомлення зверху

```
1 /* Це коментар в CSS */
2
3 .alert {
4     /* Ми додали відстань лише зверху елемента: -top */
5     margin-top: 10px;
6 }
```

Якщо ви зауважили, в дизайні зображення студентів є округлими, а у нас поки квадратні. Щоб зробити наші зображення також округлими знову звернемось до Twitter Bootstrap, а саме до [класів роботи із зображеннями](#)¹¹⁶. Скористаємось класом “img-circle”. Для цього нам не потрібно щось дописувати в CSS, а оновити тег зображення (img) і до кожного додати class=“img-circle”:

```
1 <td></td>
```

Збережіть ваш HTML файл, оновіть сторінку в браузері і впевніться, що зображення тепер округлої форми.

А тепер додамо лінію над нашим футером. Це додасть гарного візуального ефекту відділеності нижнього колонтитула. Для цього скористаємось властивістю CSS border, а також внутрішньою відстанню елемента (padding), щоб лінія не була занадто близько до тексту футера:

¹¹⁶<http://getbootstrap.com/css/#images-shapes>

main.css футер

```
1 /* знайшли елемент футера по його унікальному ідентифікатору
2   footer */
3 #footer {
4   /* задаємо верхню межу елемента: ширина лінії, тип лінії, колір
5   лінії */
6   border-top: 1px solid #dddddd;
7   /* задаємо внутрішню відстань елемента до верхнього краю */
8   padding-top: 10px;
9 }
```

Вам може бути цікаво чому я обрав саме колір “#dddddd”? Обрав я його, щоб він був такий самий, який мають внутрішні межі нашої таблички із студентами. Як я це визначив? Використовуючи чудовий інструмент Firebug, вибравши елемент в таблиці, клікнувши праву клавішу мишкої і обравши Inspect Element with Firebug:

№	Фото	Прізвище †	Ім'я	№ Білету	Дії
1		Корошт	Андрій	2123	<button>Дія ▾</button>
2		Подоба	Віталій	254	<button>Дія ▾</button>
3		Примула	Тарас	5332	<button>Дія ▾</button>

Робота з Firebug

На порядку денному останній фікс до нашої сторінки: гарно застилити меню груп. Ось що нам потрібно зробити:

- опустити меню трохи нижче;

- зробити селектор трохи вищим;
- забрати негарні межі селектора;
- зробити колір фону та тексту в селекторі сірішим, щоб краще пасував сторінці.

Давайте одразу до коду, що нам все це зробить. Ми прив'язались до двох елементів: “#group-selector” - контейнер нашого меню, а також “#group-selector select” - безпосередньо поле select:

Додаємо стилі до меню груп

```
1 /* Groups Selector */
2 #group-selector {
3   margin-top: 30px;
4 }
5 #group-selector select {
6   margin: 0;
7   width: 85%;
8   background-color: #ffffff;
9   border: 1px solid #cccccc;
10  height: 30px;
11  line-height: 30px;
12  color: #555555;
13 }
```

На домашнє завдання залишаю вам розібратись із кожною властивістю у вище наведеному куску CSS правил.

Ось кінцевий результат після усіх наших дрібних фіксів:

№	Фото	Прізвище †	Ім'я	№ Білету	Дії
1		Корост	Андрій	2123	<button>Дія</button>
2		Подоба	Віталій	254	<button>Дія</button>
3		Притула	Тарас	5332	<button>Дія</button>

Верстка Готова!

Як на мене то дуже навіть непогано! Можемо себе похвалити і нагородити чашечкою чаю чи кави та невеличкою паузою.

Репозиторій коду

На завершення глави попрактикуємося із Git репозиторієм. Закинемо нашу статичну папочку у корінь нашого репозиторію і закомітимо зміни.

Як завжди показую усе в режимі командної стрічки:

Закидуємо статичну верстку в репозиторій

```

1 # копіюємо рекурсивно папку static в папку нашого Django проекту
2 $ cp -R ~/Desktop/static/ /data/work/virtualenvs/studentsdb/src/stud\
3 entsdb/
4 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/
5
6 # додаємо нові файли та комітимо в репозиторій
7 $ git add static
8 $ git commit -a -m 'add static homepage'
9

```

```
10 # ну і на завершення не забуваємо запушити зміни на
11 # віддалений репозиторій
12 $ git push origin master
```

На github.com тепер можна переглянути свій новий коміт.

Домашнє завдання

Як бачите використовуючи існуючий фронт-енд фреймворк ми доволі просто розібралися із створенням статичної сторінки для нашого проекту. Уявіть скільки б часу зайняло це все зробити з нуля?

Протягом даної глави ми:

- базово розібралися із HTML;
- ознайомились із Twitter Bootstrap фреймворком;
- сформували вигляд головної сторінки у браузері.

На домашнє завдання ви уже отримали кілька тривіальних речей протягом глави. А тут я ще вам накидаю кілька ідей уже для складніших задач, зробивши які ви набудете неабиякого досвіду роботи із HTML та CSS.

По-перше, якщо ви цього ще не зробили, пройдіть хоча б по одному онлайн курсу по HTML та CSS. Особливо, якщо вам було важко зрозуміти деякі приклади із даної глави. Інакше, задача мінімум - детальніше ознайомитись з усіма HTML тегами та CSS властивостями, які ми використали у даній главі.

Також самостійно поекспериментуйте із Git та командною стрічкою. Особливо, якщо деякі приклади із книги вам було важко зрозуміти.

Тепер завдання середнього рівня: спробуйте самостійно зверстати подібну сторінку для списку груп:

Сервіс Обліку Студентів

Студенти Відвідування Групи

Групи

[Додати Групу](#)

#	Назва ↑	Староста	Дії
1	МтМ-21	Ячменев Олег	Дія ▾
2	МтМ-22	Віталій Подоба	Дія ▾
3	МтМ-23	Іванов Андрій	Дія ▾

1 2 [Далі](#)

© 2013 Сервіс Бази Студентів

Список Груп

для журналу відвідування:

Сервіс Обліку Студентів

Студенти Відвідування Групи

Облік Відвідування

← Жовтень 2014 →

#	Студент	Ср	Чт	Пт	Сб	Нд	Пн	Вт	Ср	Чт	Пт	Сб	Нд	Пн	Вт	Ср	Чт	Пт	Сб	Нд	Пн	Вт	Ср	Чт	Пт	Сб	Нд	Пн	Вт	Ср	Чт	Пт	Ср	Чт	Пт
1	Подоба Віталій	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>									
2	Корост Андрій	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>																						
3	Припутла Тарас	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>			

© 2014 Сервіс Обліку Студентів

Відвідування Студентів

і форму редагування студента:

Форма Студента

Усі сторінки, звичайно, мають бути окремими файлами із HTML кодом.

Якщо вищі завдання для вас видалися занадто простими, тоді тримайте ще 2. Якщо ви їх самостійно зможете виконати, то сміливо можете себе зараховувати, як мінімум, до верстальщиків середнього рівня:

- зверстати головну сторінку без Twitter Bootstrap, з нуля;
- адаптувати свою верстку з нуля під Internet Explorer, Safari, Opera, а також під мобільні пристрої (iOS, Android).

Пишіть і звітуйте ваші успіхи, а також невдачі у закриту групу підтримки. Без звітів і без роботи над домашніми завданнями, вважайте, що даремно потратили час працюючи над даною главою книги.

...

У наступній главі ми зануримось з головою в Django. Візьмемо кілька дуже важливих концепцій фреймворка як от:

- запит та відповідь, трохи про HTTP протокол;

- що таке MVC підхід?
- регулярні вирази;
- організація URL адрес аплікації;
- мова шаблонів в Django;
- в'юшки в Django;
- статичні ресурси;
- процесор контексту.

Глава буде досить об'ємною та непростою. Тому зробіть хорошу паузу та наберіться сил перед нею.

6. Динамізуємо головну сторінку

Дана глава є однією із найважливіших у книзі і однією із найскладніших. Тому хочу, щоб ви добряче підготувались і не поспішали розчаровуватись, якщо чогось не зрозумієте з першого разу. Запасіться терпінням. Можливо приайдеться перечитувати матеріал, практикуватись і пробувати це все не один раз для повного розуміння.

В даній главі ми продовжимо роботу над нашою головною сторінкою із списком студентів, розпочнемо роботу над закладкою Групи (так, це те, що ви пробували у попередній главі зверстati в якості домашнього завдання; зробили?) та зробимо наступні речі:

- перенесемо верстку головної сторінки в Django та динамізуємо більшість посилань на ній;
- список студентів зробимо динамічним, але поки дані ми самі наб'ємо у в'юшці; у наступній главі дані ми візьмемо уже з бази даних;
- розробимо структуру адрес нашої аплікації: закладка Групи, Відвідування, форми Додавання, Редагування та Видалення Студента та Групи;
- реалізуємо сторінку із списком груп (закладка Групи) уникнувши дубляжу в повторюваних елементах на сторінках нашої аплікації.

Протягом реалізації вище наведеного функціоналу ми з вами розберемось із наступними концепціями:

- HTTP протокол;
- MVC підхід у веб-розробці, зокрема в Django;
- *Регулярні Вирази*, зокрема їхнє використання у Django;
- Django аплікація;

- Django URL диспетчер;
- Django в'юшки;
- мова Django шаблонів, унаслідування;
- статичні ресурси в Django;
- процесор контексту.

Як бачите роботи в нас протягом даної глави буде дуже багато. Тому запасіться терпінням, часом, горнятком чаю (чи іншого вашого улюбленого напою) і поїхали!

HTTP протокол

Перед тим як братись до написання наших перших Django сторінок (views - в'юшки) нам варто коротенько розібратись із нутрощами протоколу HTTP. Це допоможе нам краще зрозуміти принцип роботи різних компонент Django, а також зв'язок між ними.

HTTP протокол - це протокол передачі HTML документів між клієнтом (веб-переглядачем) та сервером. Звичайно він також використовується для передачі і інших типів документів та файлів (CSS, Javascript, зображення і т.д.), але HTML документи - це найпоширеніше використання даного протоколу.

Кожна сторінка (ресурс) в інтернеті унікально ідентифікується за допомогою адреси: URL (Uniform Resource Locator - уніфікований вказівник на ресурс).

Кожного разу, коли ми клікаємо на посилання на веб-сторінці або набираємо адресу веб-сайту у стрічці адреси веб-браузера, ми з вами виконуємо *запит* (request) на сервер. Сервер отримує запит, обробляє його, а потім готує та відправляє нам *відповідь* (response).

Крім того запити бувають різних типів. Нас з вами цікавлять лише два найпоширеніші типи запиту:

- GET: використовується для запитів на сервер по інформацію; дефолтний тип запиту; зазвичай виконується при навігації по лінках, при користуванні стрічкою адреси веб-браузера;

- POST: використовується для запитів на сервер із потребою доставити дані та оновити дані на сервері; використовується для відправки даних форм на сервер.

Кожен запит складається із рядочка статусу, заголовків та тіла запиту:

```
POST /subscribe/post HTTP/1.1
Host: vitaliyopoda.us6.list-manage1.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:33.0) Gecko/20100101 Firefox/33.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://vitaliyopoda.us6.list-manage.com/subscribe/post?u=8a58628378&id=f8a8bd7c3e
Connection: keep-alive
```

Формат HTTP запиту

Перший рядок називається рядком статусу. Він містить: тип запиту (у нашому випадку це POST), адреса запиту (/subscribe/post), версія HTTP протоколу (HTTP/1.1). Наступні рядки є заголовками запиту і містять різного ряду мета-інформацію про:

- домен веб-сайту;
- версії браузера та операційної системи;
- інформацію про підходящу мову, кодування для веб-браузера;
- і багато інших речей, взалежності від типу запита та вашого веб-переглядача.

Після заголовків йде порожній рядок, за яким слідує тіло запиту. Воно присутнє у GET запитах, якщо в URL адресі є додаткові параметри (інформація в адресі, що йде після символу знака питання (?), наприклад адреса “<http://example.com/?key1=value1>” містить параметр “key1=value1”).

Для тих, хто хоче достеменно розібратись із форматом запиту, в [цьому документі](#)¹¹⁷ детально описано формат запиту.

Відповідь має подібний формат:

¹¹⁷ <http://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html>

```

HTTP/1.1 200 OK
Server: nginx
Content-Type: text/html; charset=UTF-8
X-UA-Compatible: IE=edge,chrome=1
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Content-Encoding: gzip
Vary: Accept-Encoding
Date: Mon, 10 Nov 2014 15:17:14 GMT
Content-Length: 8355
Connection: keep-alive
Set-Cookie: AVESTA_ENVIRONMENT=prod; path=/
PHPSESSID=d9cdr4nava40npiqppak4n7dk0; path=/

```

Формат HTTP відповіді

Як бачите перший рядок статусу трохи відрізняється від рядка в запиті і містить: версію протоколу (HTTP/1.1), код статусу відповіді (200) та статус відповіді у більш “user-friendly” форматі (OK).

Далі йдуть заголовки, що містять зазвичай інформацію про:

- деталі сервера, який обробив наш запит;
- дані сторінки, як от дата модифікації, термін дії;
- заголовки щодо кешування сторінки;
- довжина даних тіла відповіді;
- тип зв’язку із сервером;
- тип документу та тип кодування, якщо присутнє;
- та інші мета дані взалежності від типу сервера та запиту.

Після заголовків йде порожній рядок, за яким слідує тіло відповіді. Зазвичай це HTML текст.

[Статуси відповідей¹¹⁸](#) поділяються на 5 категорій:

- 1xx: інформаційні;
- 2xx: успішні операції;
- 3xx: переадресації сторінки (коли ви робите запит за однією сторінкою, а вас сервер перенаправляє автоматично на іншу; за це відповідає заголовок Location);
- 4xx: клієнтські помилки;
- 5xx: помилки на стороні сервера.

¹¹⁸ <http://bit.ly/vphttpstatus>

Найбільш поширеними статусами є:

- 200 OK: відповідь успішна;
- 301 Moved Permanently: постійний редірект;
- 401 Unauthorized: немає доступу;
- 404 Not Found: сторінка не знайдена;
- 500 Internal Server Error: внутрішня помилка сервера.

Протокол HTTP є “stateless” (протокол без стану). Це означає, що між вашим попереднім і наступним запитами на сервер немає ніякого зв’язку. Щоб обійти це обмеження і, наприклад, мати можливість логувати користувача на довший термін, ніж один запит, програмісти на вищому рівні обходять це обмеження і реалізовують цей додатковий стан (зв’язок між різними запитами від однієї людини) інструментами, що доступні на стороні сервера (користувальські сесії, додаткові параметри) та клієнта (cookies, автентифікаційні хідера).

Cookies

На завершення даної секції розглянемо поняття **Cookies**¹¹⁹ (з англ. - печеньки, куки).

Cookies - це невеликий кусок інформації, що зберігається у вашому веб-переглядачі. Його встановлює веб-браузер на прохання сервера. Дані інформація зберігається під певним ключом (іменем куки), певний період часу (або протягом браузерної сесії, тобто після перезапуску веб-переглядача така кука пропаде) та для певного домену і підшляху веб-сторінки.

Встановлені куки браузер кожного разу відсилає на сервер. Таким чином Cookies зазвичай використовуються, щоб ідентифікувати користувача. З їх допомогою реалізуються користувальські сесії, налаштування під користувача

¹¹⁹ <http://bit.ly/vphttpcookies>

та будь-який функціонал пов'язаний із збором інформації щодо поточного користувача.

У нашому проекті ми скористаємось куками, коли будемо реалізували меню Груп. Тоді і детальніше розберемось як вони працюють.

Цієї інформації про HTTP протокол нам буде достатньо, щоб в подальшому краще зрозуміти принцип роботи Django сервера, диспетчера URL адрес та самих сторінок - в'юшок.

Що таке MVC?

Гляньте на наступний php файл із Wordpress CMS:

```

96:     > printf( __( 'Your account has been activated. You may now <a href="%1$s">log in</a> to the site using your chosen username of %2$s' ), $user_login, $user_login );
97: } else {
98:     printf( __( 'Your site at <a href="%1$s">%2$s</a> is active. You may now log in to your site using your chosen username of %2$s' ), $url, $site );
99: }
100: echo '</p>';
101: } else {
102:     ?>
103:     <h2><?php _e('An error occurred during the activation'); ?></h2>-
104:     <?php
105:     echo '<p>' . $result->get_error_message() . '</p>-
106:     ?>
107: } else {
108:     extract($result);
109:     $url = get_blogaddress_by_id( (int) $blog_id );
110:     $user = get_userdata( (int) $user_id );
111:     ?>
112:     <h2><?php _e('Your account is now active!'); ?></h2>-
113:     <div id="signup-welcome">
114:         <p><?php _e('Username:'); ?></p>-
115:         <p><?php _e('Password:'); ?></p>-
116:     </div>-
117:     <?php if ( $url != network_home_url('', 'http') ) : ?>-
118:         <?php $class = 'view'; ?><?php printf( __( 'Your account is now activated. <a href="%1$s">View your site</a> or <a href="%2$s">Log in</a>' ), $url, $login ); ?>
119:     <?php else : ?>-
120:         <?php $class = 'view'; ?><?php printf( __( 'Your account is now activated. <a href="%1$s">Log in</a> or go back to the <a href="%2$s">homepage</a>' ), $login, $url ); ?>
121:     <?php endif; ?>
122:     </div>-
123:     <script type="text/javascript">-
124:     var key_input = document.getElementById('key');-
125:     ?>
126:     ?>
127:     </script>-
128: 
```

wp-activate.php модуль із Wordpress CMS

PHP код йде в перемішку із HTML кодом. Частина HTML коду є напряму вкладена у файл, а частина тексту виводиться PHP функцією “printf”. Крім того в PHP коді бачимо виклики функцій та роботу із об'єктами. Частина з цих викликів працює із базою даних.

А тепер уявіть, що над цим файлом працюють одночасно верстальщик та PHP програміст. Якщо програміст ще розбереться із ним, то верстальщик може легко нарубати дров випадково поламавши одну із директив PHP коду.

Щоб уникати таких проблем та полегшити собі життя, програмісти придумали шаблон проектування аплікації - **MVC¹²⁰**. Цей підхід розділяє код програми на три різні частини:

- дані програми: також називаємо цю частину Моделлю (Model); тут ми зберігаємо дані аплікації;
- представлення даних: частина коду, яка відповідає за представлення даних користувачу; також називаємо дану частину коду В'юшкою (View);
- логіка аплікації: або Контроллер (Controller); дана компонента обробляє запити від користувача, маніпулює даними і передає їх у потрібному форматі компоненті View.

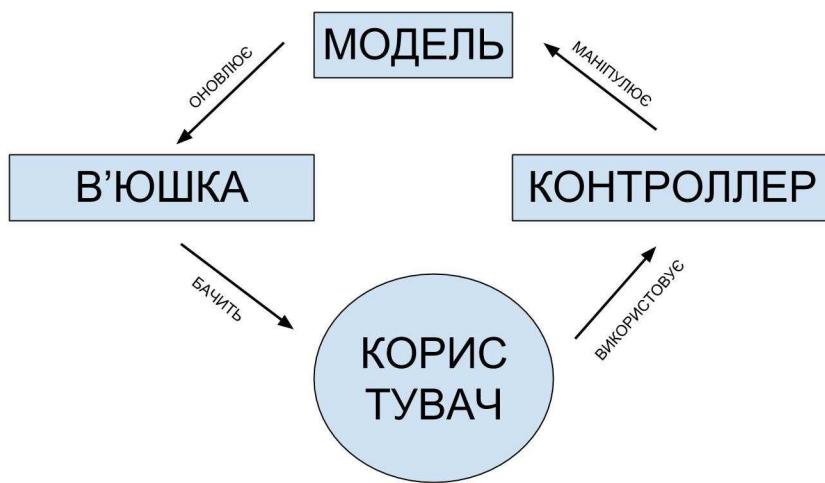


Схема MVC: Користувач використовує компоненту Controller для обробки своїх записів, далі Controller при потребі оновлює дані (Model), далі оновлюється сторінка для користувача базуючись на останніх змінах в даних (View)

Таким чином в одному проекті можемо мати три максимально незалежні компоненти, над якими можуть працювати три різні люди не заважаючи один одному. При потребі змін у одну компоненту, зміни у інші будуть мінімальні. Це і є основні задачі MVC підходу при проектуванні програм.

¹²⁰<https://ru.wikipedia.org/wiki/Model-View-Controller>

Конкретно у веб-аплікаціях представлення даних користувачу відбувається з допомогою мови HTML. Відповідно верстальщик може працювати із своєю частиною роботи по мінімуму вникаючи у роботу програміста, адже програміст більшість свого часу буде проводити над Контроллером та Даними програми.

MVC в Django - це MTV

У Django MVC компоненти реалізовано дещо по-іншому.

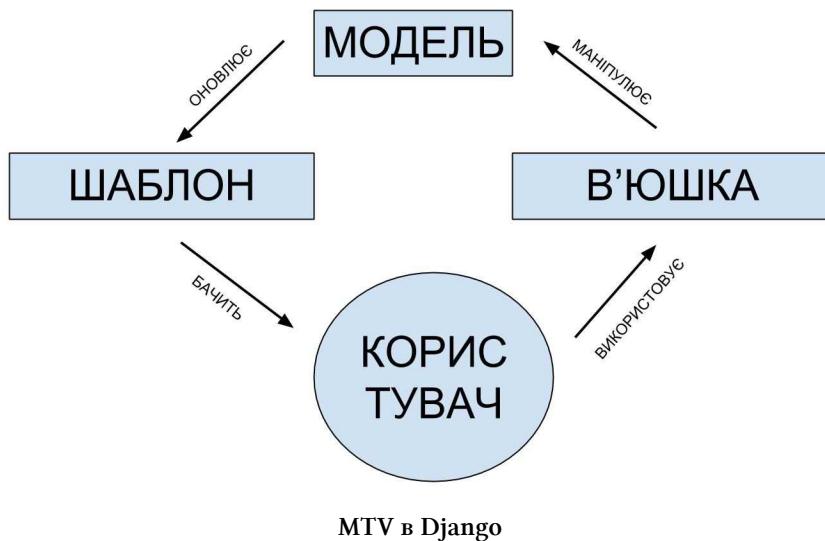
Компоненту Даних (Моделі) виконують Django Моделі, які є Python класами з набором відповідних полів. Проте інші компоненти мають дещо інші назви у фреймворку Django.

Кінцевий HTML код генерується Django шаблонами, які надають розробнику набір динамічних тегів та змінних. Тобто MVC компоненту View в Django будемо називати Шаблоном (Template).

А от усією логікою в Django аплікації стосовно обробки запиту та формування відповіді клієнту займається в'юшка. Функція (або клас), яка має доступ до запиту та до бази даних. Дані функція зазвичай повертає HTML код згенерований Шаблоном. Таким чином MVC компонента Controller в контексті Django буде називатись View.

Таким чином MVC в Django адаптовано під MTV (Model Template View). Це варто запам'ятати.

Насправді чіткої відповідності MVC в Django немає. Деято включає Django URL диспетчер разом із View в MVC Controller також.



А загалом у Django добре продуманий підхід із реалізацією логіки, даних та представлення даних таким чином, щоб ви, навіть будучи початківцем, не наламали дров і не вмістили логіку у представлення, а дані у логіку.

Єдине зауваження, яке я персонально маю до MVC в Django, є те, що згідно кращих практик Django розробки, потрібно частину логіки вкладати у класи моделей. Це суперечить MVC підходу, адже дані ніколи не повинні знати про те, як, хто і коли використовуватиме їх. Це все повинно бути частиною контроллера... В будь-якому випадку маєте їжу для роздумів...

Регулярні вирази

Це остання теоретична секція даної глави перед тим, як перейти до практики. Вона просто необхідна, щоб далі розуміти як ми з вами будемо формувати URL структуру нашого проекту.

В Django є URL диспетчер (інструмент для з'єднання URL адрес на веб-сайті із їх обробниками в коді), який побудований з інтенсивним використанням

Регулярних Виразів.

Регулярні Вирази¹²¹ - це формальна мова для пошуку та маніпуляцій з підстрічками у тексті. Вона базується на використанні так званих метасимволів (wildcard, або ще їх називають джокерами). Регулярний Вираз - це стрічка (також називають її маска, шаблон), яка складається із символів та метасимволів, які задають правило для пошуку під-стрічки у тексті.

Думаю ви могли вже зустрічатись із регулярним виразом при пошуку файлів у вашій операційній системі. Наприклад, шукаючи усі файли з розширенням .exe, ви вводили у поле пошуку щось подібне на “*.exe”. В даному випадку символ зірочки * є метасимволом, який означав “будь-яка кількість будь-яких символів” у назві файлу перед закінченням, що мало бути рівним стрічці “.exe”.

Так от, у вище наведеному прикладі і заложена вся сутність регулярних виразів: ідентифікувати частину тексту, яка задовільняє певним правилам. А для того, щоб знати як можна формувати дані правила, давайте розберемо основні метасимволи, що використовуються в регулярних виразах.

Для тестування усіх наступних прикладів рекомендую скористатись онлайн сервісом [regex101¹²²](http://regex101.com). Вибрали опцію “FLAVOR” рівне Python, починайте вводити ваш вираз у поле REGULAR EXPRESSION, а текст для тестування у поле TEST STRING.

Метасимволи, що пригодяться нам в роботі над проектом

Символи	Опис	Вираз	Результат
^	Початок стрічки.	^abc	перше “abc” в “abc-abc”
\$	Кінець стрічки.	abc\$	друге “abc” в “abc-abc”
[abc]	Будь-який один символ в множині.	[ak]	“a” в “abc”
[^abc]	Будь-який символ поза даною множиною.	[ak]	“b”, “c” в “abc”
[0-9]	Будь-який елемент в проміжку.	[0-9]	“5” в “or 5 new”
	Будь-який один елемент в списку.	(e is at)	“e”, “is” в “he is”
\d	Будь-який цифровий символ.	\d	“6” в “at6b”
\D	Будь-який не цифровий символ.	\D	“a”, “t”, “b” в “at6b”

¹²¹<http://bit.ly/vpregex>

¹²²<http://regex101.com/>

Метасимволи, що пригодяться нам в роботі над проектом

Символи	Опис	Вираз	Результат
\w	Будь-який символ, що використовується в словах.	\w	“I”, “D”, “3” в “ID.3”
\W	Будь-який символ, що не використовується у словах.	\W	” ”, “.” в “ID A1.3”
\s	Будь-який символ пробілу.	\w\s	“D ” в “ID A1.3”
\S	Будь-який символ, що не є пробілом.	\s\S	” _ ” в “int __ctr”
.	Будь-який символ, окрім перевodu рядка.	a.e	“ave” в “nave”
*	Попередній елемент може повторюватись 0 або більше разів.	\d*\.\d	“.0”, “19.9”, “219.9”
+	Попередній елемент може повторюватись 1 або більше разів.	“be+”	“bee” в “been”
?	Попередній елемент може повторюватись 0 або 1 раз.	“rai?n”	“ran”, “rain”
{n}	Попередній елемент може повторюватись рівно n разів.	”,\d{3}”	”,043” в “1,043.6”
{n,m}	Попередній елемент може повторюватись від n до m разів.	“\d{3,5}”	“166”, “17668”
()	Неіменована група. Запам'ятує внутрішній вираз під порядковим номером починаючи з 1. Під нулем запам'ятується весь вираз.	a(\S+)b	“_.-“ в “a_.-b”
(?P<sid>)	Іменована група. Замап'ятує внутрішній вираз під вказаним ім'ям. В даному випадку “sid”.	a(\S+)b	“_.-“ в “a_.-b”

Крім того при тестах регулярних виразів можна застосовувати так звані флагки, щоб наприклад тестиувати незалежно від реєстру символів, або тестиувати багатострічковий текст. Ми флагки не розглядаємо, адже використовуватимемо регулярні вирази у Django URL диспетчері, для якого нам вони не пригодяться. На самостійне опрацювання поекспериментуйте

з Python модулем регулярних виразів: re.

Найскладнішим регулярним виразом, який ми будемо з вами використовувати в процесі роботи над наших проектом буде: “`^students/(?P<sid>[0-9]+)/edit/$`”. Якщо ви його в даний момент не до кінця розумієте, тоді зверніться будь-ласка до відео уроку про регулярні вирази, що йде разом із Рекомендованим Пакетом даної книги, а також далі поекспериментуйте із онлайн тестером регулярних виразів. Це вам допоможе розібратись із базовими речима і далі без проблем розуміти нашу роботу в urls.py модулі Django проекта.

Створюємо Django аплікацію

Тепер ми озброєні, щоб рухатись далі до практичної частини цієї глави. А розпочнемо ми із створення Django аплікації всередині нашого Django проекту.

Django Аплікація - це Python пакет, який містить окремий аспект чи автономний функціонал Django проекту. Згідно кращих практик кожна аплікація повинна містити окремий функціонал Django проекту. Наприклад, якщо у вас на сайті є блог, розділ новин, розділ подій, розділ магазину, тоді кожен із даних розділів повинен бути реалізованим як окрема Django аплікація.

У дану аплікацію ми складатимемо наші моделі, в'юшки, шаблони та, зрештою, статичні ресурси. А під'еднуватимемо усі дані ресурси через налаштування самого проекту.

Створюємо Django аплікацію students в корені нашого репозиторію

```
1 # заходимо у корінь нашого віртуального середовища та активуємо його
2 $ cd /data/work/virtualenvs/studentsdb/
3 $ source bin/activate
4
5 # заходимо в корінь нашого репозиторію та створюємо Django аплікацію
6 # командою startapp
7 (studentsdb)$ cd src/studentsdb/
8 (studentsdb)$ python manage.py startapp students
```

Для Windows Ніньдзя: усі папочки, що на Linux та Mac OS називаються “bin”, на Windows системі зазвичай називаються “Scripts”. Майте на увазі!

Тепер в кореневій папочці репозиторію з'явилась ще одна підпапка під назвою “students”. У ній ми можемо побачити наступні папки та файли:

- `__init__.py`: порожній Python модуль, щоб наша папка була Python пакетом;
- `admin.py`: тут будуть лежати реєстрації наших моделей, щоб вони з'явилися в Django адмінці;
- `migrations`: папочка міститиме міграції моделей;
- `models.py`: модуль із майбутніми моделями нашого проекту;
- `tests.py`: тут будуть наші юніт тести;
- `views.py`: модуль вмістилище для наших в'юшок; цієї глави ми добраче попрацюємо із ним.

Тепер, щоб наш Django проект побачив новостворену аплікацію, потрібно додати її у список заінсталюваних аплікацій в модулі `settings.py` в папці проекту:

Додаємо students аплікацію до інсталюваних аплікацій в settings.py

```
1 INSTALLED_APPS = (
2     'django.contrib.admin',
3     'django.contrib.auth',
4     'django.contrib.contenttypes',
5     'django.contrib.sessions',
6     'django.contrib.messages',
7     'django.contrib.staticfiles',
8     'students',
9 )
```

Бачите у 8му рядку ми додали стрічку ‘students’ в змінну-список INSTALLED_APPS. Тепер Django буде читувало та ініціалізувало моделі та інші модулі із аплікації students.

Зміни в Git

Перед тим як рухатись далі, закомітимо нові зміни в репозиторій коду:

Комітимо зміни в репозиторій

```
1 # знаходячись у кореневій папці нашого репозиторію перевіряємо
2 # стан справ
3 $ git status
4 # On branch master
5 # Changed but not updated:
6 #   (use "git add <file>..." to update what will be committed)
7 #   (use "git checkout -- <file>..." to discard changes in working d\
8  irectory)
9 #
10 # modified:   studentsdb/settings.py
11 #
12 # Untracked files:
13 #   (use "git add <file>..." to include in what will be committed)
```

```
14 #
15 # students/
16 no changes added to commit (use "git add" and/or "git commit -a")
```

Бачимо, що змінений модуль settings.py. А також з'явилася нова папочка "students/", яка ще не є в репозиторії.

А тепер, щоб побачити зміни в існуючих файлах скористуємось командою git diff:

Переглядаємо зміни перед комітом

```
1 $ git diff
2 diff --git a/studentsdb/settings.py b/studentsdb/settings.py
3 index 11a6259..d06d916 100644
4 --- a/studentsdb/settings.py
5 +++ b/studentsdb/settings.py
6 @@ -36,6 +36,7 @@ INSTALLED_APPS = (
7     'django.contrib.sessions',
8     'django.contrib.messages',
9     'django.contrib.staticfiles',
10    + 'students',
11 )
12
13 MIDDLEWARE_CLASSES = (
```

Даний вивід показує нам, які саме рядочки змінились у наших існуючих файлах. Знак мінус - рядочок старий, знак плюс - рядочок новий.

Візьміть собі за звичку перед кожним комітом користуватись командою "git status" та "git diff", щоб бути впевненим у тому, які саме зміни ви закидаєте в репозиторій. Це дозволить уникати багатьох дрібних проблем.

Робимо коміт в локальний репозиторій та пушимо зміни на віддалений репозиторій:

Закидаємо останні зміни в репозиторій

```
1 # додаємо нові папки і файли до індекса (ще його називаємо стейджинг,
2 # staging); будучи в корені нашого репозиторія:
3 $ git add students/
4
5 # комітимо усі зміни, включно з існуючими файлами (опція -a):
6 $ git commit -a -m 'add students app to our project'
7
8 # закидуємо останні коміти у віддалений репозиторій
9 $ git push origin master
10
11 # якщо ви клонували репозиторій, тоді origin і master для вас
12 # запам'ятатись і при наступних пушах саме master гілки не потрібно
13 # вказувати ні назви віддаленого репозиторію, ні назви гілки:
14 $ git push
```

Список студентів в Django

В даній секції ми дуже класно попрацюємо над шаблоном із списком студентів та розберемось із базисом Django в'юшок та URL конфігурацією проекту.

Демо в'юшка

Перед тим, як почнемо переносити наш статичний файл у новостворену аплікацію та підключати його до в'юшки, що оброблятиме головну сторінку, поекспериментуємо із простішою версією сторінки. В той же час трохи розберемось із тим, що таке в'юшка в Django і яке її основне призначення.

Давайте відкриємо у редакторі модуль `views.py` всередині нашої аплікації `students` та додамо код для надзвичайно простої Django в'юшки:

Наша перша Django в'юшка, файл views.py

```
1 from django.shortcuts import render
2 from django.http import HttpResponseRedirect
3
4 def students_list(request):
5     return HttpResponseRedirect('<h1>Hello World</h1>')
```

На що варто звернути увагу?

- 2ий рядок: я імпортував клас HttpResponseRedirect з пакету django.http;
- 4ий: в'юшка в Django є функцією (як у нашому прикладі) або класом (про це пізніше); функція приймає як мінімум один обов'язковий аргумент - request: запит - об'єкт сформований для нас фреймворком;
- 5ий: функція в'юшки віддає новостворений об'єкт класу HttpResponseRedirect, якому ми, в свою чергу, передали HTML код як стрічку.

Таким чином маємо запам'ятати 3 важливі речі про Django в'юшку:

- це функція, яка отримує як мінімум один аргумент "request";
- завжди повертає об'єкт типу HttpResponseRedirect;
- або викидає помилку:

Приклад в'юшки, яка викидає помилку

```
1 from django.http import Http404
2 from django.http import HttpResponseRedirect
3
4 def students_list(request, sid):
5     try:
6         sid = int(sid)
7     except ValueError:
8         raise Http404
9     else:
10        return HttpResponseRedirect('<h1>Hello World</h1>')
```

У цьому прикладі наша в'юшка також приймає другий аргумент sid (коли це відбувається ми розберемо пізніше) і пробує конвертувати його до цілого числа. Якщо це не вдається і виникає помилка типу “`ValueError`”, тоді ми спеціально викидаємо іншу помилку типу “`Http404`”, що означає, що ресурс не знайдено.

Якщо вище наведений приклад вам не повністю зрозумілий, тоді будь-ласка перегляньте документацію з мови Python, а саме: робота з помилками (Exceptions).

Об'єкт `request`, що отримує в'юшка має масу методів, атрибутів та інформації всередині. Починаючи від стандартних заголовків згідно протоколу HTTP і закінчуючи змінними віртуального середовища та інформацією про Django:

- `request.path`: шлях запиту;
- `request.GET`: дані, що прийшли частиною GET параметрів;
- `request.POST`: дані, що прийшли в тілі поста форми із браузера;
- `request.method`: тип запиту, напр. GET або POST;
- `request.COOKIES`: славнозвісні Cookies, які ми з вами оглянули в попередніх секціях даної глави;
- `request.FILES`: файли, що прийшли до нас із форми, із полів типу file;
- `request.META`: заголовки запиту згідно HTTP протоколу;
- `request.user`: залогований Django користувач;
- `request.session`: сесія користувача;
- і ще багато інших...

Наша в'юшка в Django має кілька різних опцій, щоб повернути відповідь у браузер.

Одним із найдовших варіантів буде завантажити і відрендерити (від англ. `render` - надавати, виконувати) HTML код із Django шаблону, створити для цього так званий контекст і в кінці кінців повернути із функції новостворений об'єкт `HttpResponse`:

Готуєм відповідь із Django шаблона

```
1 from django.http import HttpResponseRedirect  
2 from django.template import RequestContext, loader  
3  
4 def students_list(request):  
5     template = loader.get_template('demo.html')  
6     context = RequestContext(request, {})  
7     return HttpResponseRedirect(template.render(context))
```

А тепер порівняйте як можна зробити ту ж дію використовуючи єдину функцію, так званий Django shortcut, під назвою render із пакета django.shortcuts:

Готуємо відповідь із Django шаблона використовуючи функцію render

```
1 from django.shortcuts import render  
2  
3 def students_list(request):  
4     return render(request, 'demo.html', {})
```

В деталі вдаватись не будемо. Усі ці варіанти ми використаємо при реалізації наших сторінок для проекту. На даний момент просто запам'ятайте, що реалізувати в'юшку, яка повертає об'єкт HttpResponseRedirect можна багатьма різними способами.

Більше про об'єкти запиту та відповіді в Django можете прочитати [тут¹²³](#).

URL конфігурація

Тепер ми маємо демо Django в'юшку під назвою `students_list`, яка повертає у браузер відповідь із простим HTML кодом: заголовком першого рівня.

Але ще й досі ми не повідомили Django про існування даної функції `students_list`. Щоб це зробити нам треба скористатись модулем `urls.py`, що знаходиться в пакеті проекту:

¹²³<http://djbook.ru/rel1.4/ref/request-response.html>

urls.py: додаємо students_list як в'юшку для кореня нашої аплікації

```
1 from django.conf.urls import patterns, include, url
2 from django.contrib import admin
3
4 urlpatterns = patterns('',
5     # Students urls
6     url(r'^$', 'students.views.students_list', name='home'),
7
8     url(r'^admin/', include(admin.site.urls)),
9 )
```

Приблизно таким чином повинен виглядати ваш urls.py модуль:

- 1-2 рядки: імпортуємо необхідні модулі;
- 4ий: змінну urlpatterns для нас вичитає Django URL диспетчер і ініціалізує згідно даних правил структуру URL адрес веб-аплікації; функція patterns() правильно зформатує список переданих їй шаблонів; порожня стрічка як перший аргумент функція - це префікс¹²⁴;
- 6ий: виклик функції url¹²⁵; вона містить деталі про один тип адрес на нашему сайті;
- 8ий: всі запити, що будуть йти на розділ нашої аплікації під адресою “admin/” будуть оброблятись Django аплікацією “django.contrib.admin”.

Функція url приймає наступні аргументи:

- *регулярний вираз*: визначає правило для пошуку URL адреси, що відповідає даному запиту;
- *посилання на в'юшку*: може бути функція, або посилання на функцію у вигляді стрічки із шляхом до назви функції; дана в'юшка запуститься, якщо регулярний вираз задовільнятиме URL адресі запиту;
- *name*: назва в'юшки; цей аргумент нам пізніше пригодиться в шаблонах, щоб будувати динамічно URL адреси для лінків на веб-сторінці.

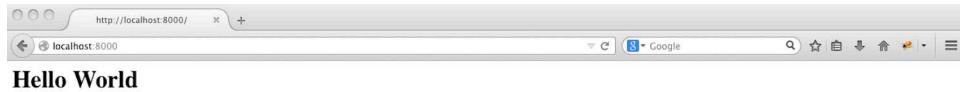
¹²⁴<http://djbook.ru/rel1.4/topics/http/urls.html#patterns>

¹²⁵<http://djbook.ru/rel1.4/topics/http/urls.html#url>

Також функція url має ще параметри kwargs та prefix, але ми їх залишимо на потім.

Отже, першим патерном у нашій змінній urlpatterns увійшов патерн (синонім до слова шаблон) із регулярним виразом “^\$”. Це означає порожню стрічку. Беручи до уваги, що домен та слеш, що йому слідує, не включаються в шлях для перевірки URL шаблонів, даний патерн задовільняє усім запитам, що йдуть на корінь нашої веб-аплікації. При запиті на корінь сайту спрацьовуватиме в'юшка за адресою: students.views.students_list. Це функція, яку ми щойно із вами реалізували, наша демо в'юшка. Цьому патерну ми дали назву “home”.

Таким чином ми підключили нашу демо в'юшку в якості обробника кореневої сторінки нашої веб-аплікації. Тепер, якщо оновите сторінку в браузері під адресою “<http://localhost:8000>”, то замість дефолтної сторінки Django, отримаєте наступну сторінку:



Демо сторінка

Переносимо HTML файл із списком студентів

Дійшли нарешті до моменту, коли можна зайнятись нашим HTML файликом із версткою списку студентів!

Для цього створимо ‘templates’ підпапку в папці нашої аплікації ‘students’. I покладемо туди наш index.html переназвавши його в ‘students_list.html’.

Переносимо index.html в Django аплікацію

```
1 # заходимо в папочку аплікації, створюємо підпапку 'templates' та
2 # у ній ще одну підпапку - 'students'
3 $ cd students
4 $ mkdir -p templates/students
5
6 # копіюємо index.html в templates/students
7 $ cp ../static/index.html templates/students/students_list.html
```

Після цього наш шаблон уже доступний для завантаження із нашої в'юшки. Нам ніяк додатково не потрібно повідомляти Django про нову папочку 'templates', адже фреймворк по-замовчуванню аналізує усі заіnstальовані аплікації на наявність підпапки 'templates'. Якщо така папка існує, тоді вона автоматично додається в реєстр.

Також зауважте, що ми додали наш шаблон не напряму в папку 'templates', а створили ще одну внутрішню папку 'students' і уже в неї поклали наш шаблон. Це для того, щоб назви шаблонів не перетиналися із уже існуючими шаблонами доступними як в самому Django, так і в сторонніх аплікаціях. Таким чином, щоб завантажити наш шаблон із в'юшки, потрібно буде вказувати "students/students_list.html" шлях, а не просто "students_list.html". Так ми гарантуємо, що наше ім'я файлу не перетнеться із іншими.

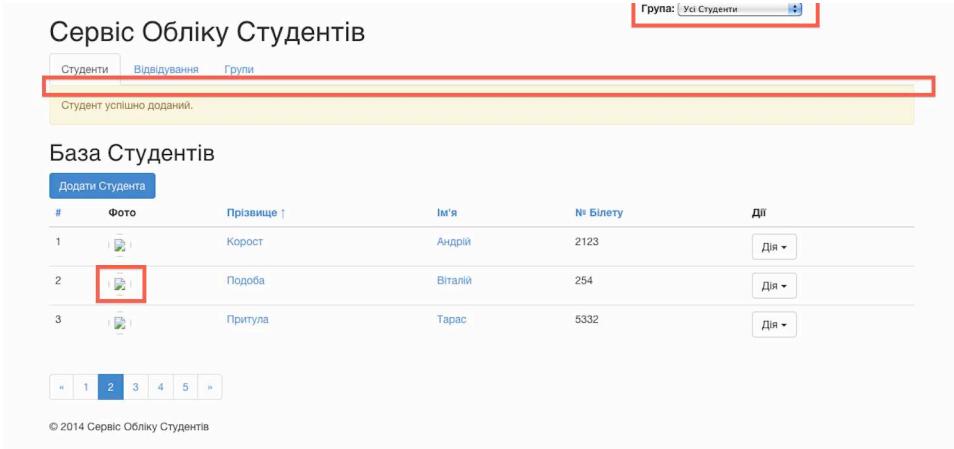
В нашу папочку 'templates' всередині аплікації ми складатимемо більшість шаблонів щодо нашого проекту. Проте, коли ми захочемо змінити стандартні шаблони Django (напр. стандартну сторінку помилки Not Found 404 або форму логування) або шаблони, що прийшли із інших існуючих Django аплікацій, тоді ми використовуватимемо папку 'templates' всередині нашого проекту. Таку папку нам потрібно буде окремо зареєструвати в модулі settings.py. Такий розподіл потрібен лише для логічного групування шаблонів від взалежності їх призначення та принадлежності.

Тепер, щоб побачити наш шаблон у якості головної сторінки аплікації, потрібно оновити код в'юшки `students_list`, щоб вона завантажувала саме цей шаблон, а не просто виводила напряму переданий HTML код:

`students_list` в'юшка віддає `students_list.html` шаблон

```
1 from django.shortcuts import render
2 from django.http import HttpResponse
3
4 def students_list(request):
5     return render(request, 'students/students_list.html', {})
```

Але бачите, є кілька нюансів, якщо тепер перезавантажите сторінку “localhost:8000” у веб-переглядачі. Зображення студентів не показуються, відстань над статусним повідомленням зникла, стилі меню груп також пропали. Все це зумовлено тим, що ми ще не перенесли статичні ресурси:



The screenshot shows a web application titled "Сервіс Обліку Студентів". At the top, there is a search bar labeled "Група:" with the value "Усі Студенти". Below the header, there are three navigation tabs: "Студенти" (selected), "Відвідування", and "Групи". A success message "Студент успішно доданий." is displayed. The main content area is titled "База Студентів" and contains a table with student data. The table has columns: "#", "Фото", "Прізвище", "Ім'я", "№ Білету", and "Дії". The data rows are:

#	Фото	Прізвище	Ім'я	№ Білету	Дії
1		Корост	Андрій	2123	<button>Дія</button>
2		Подоба	Віталій	254	<button>Дія</button>
3		Припутла	Тарас	5332	<button>Дія</button>

At the bottom, there is a page navigation bar with links "«", "1", "2" (highlighted in blue), "3", "4", "5", and "»". A copyright notice "© 2014 Сервіс Обліку Студентів" is at the very bottom.

Наш шаблон у Django, але поки без підключених статичних ресурсів

URL структура веб-аплікації

Проте перед тим, як перейти до підключення статичних ресурсів у Django, давайте спочатку сплануємо структуру веб-адрес нашої аплікації та “захард-кодимо” (hardcode - вписати статичний текст) дані адреси у лінки на сторінці

із списком студентів. В наступних секціях ми їх класно динамізуємо уже маючи набиті URL патерни в модулі urls.py в пакеті проекту.

Отже, маємо головну сторінку, яка є списком студентів і обслуговуємо ми її під шляхом '/'. Проте для подальших дій із студентами нам прийдеться мати окрему під-секцію веб-адрес. Нехай це буде '/students'. Ось список дій та відповідних шаблонів адрес, що ми використовуватимемо для кожної дії:

- список студентів: '/';
- форма додавання студента: '/students/add';
- форма редагування студента: '/students/125/edit', "125" - це приклад унікального ідентифікатора студента в базі даних; нам потрібен цей ідентифікатор, щоб знати якого саме студента потрібно редагувати;
- видалити студента: '/students/125/delete'.

Подібна структура адрес буде і для закладки Групи, щоб користувач аплікації міг працювати із створенням, редагуванням та видаленням існуючих груп:

- список груп: '/groups';
- форма додавання групи: '/groups/add';
- форма редагування групи: '/groups/213/edit';
- видалити групу: '/groups/213/delete'.

Ну і на закінчення маємо секцію Відвідування, для якої виберемо адресу '/journal' (з англ. - журнал):

- журнал відвідування студентів: '/journal';
- журнал відвідування конкретного студента: '/journal/125';
- адреса для оновлення журналу відвідування: '/journal/update'.

Тепер давайте пройдемось по лінках у нашій поки зовсім статичній сторінці і оновимо лінки в навігації, списку студентів та Діях над студентами. Зробимо це так, щоб дані лінки відображали щойно підготовану структуру веб-адрес як наведено вище:

Вибірково наведено частини файлу students_list.html з оновленими посиланнями

```
1 <!-- Тут ми оновили лінки головної навігації: Студенти,  
2     Відвідування, Групи; атрибути "href" всередині "а" тегів -->  
3 <!-- Start subheader -->  
4 <div class="row" id="sub-header">  
5     <div class="col-xs-12">  
6         <ul class="nav nav-tabs" role="tablist">  
7             <li role="presentation" class="active"><a href="/">Студент\  
8 и</a></li>  
9             <li role="presentation"><a href="/journal">Відвідування</a\>  
10 ></li>  
11             <li role="presentation"><a href="/groups">Групи</a></li>  
12         </ul>  
13     </div>  
14 </div>  
15 <!-- End subheader -->  
16  
17 <!-- А тут оновлена адреса кнопки Додати Студента -->  
18     <h2>База Студентів</h2>  
19     <a href="/students/add" class="btn btn-primary">Додати Студе\>  
20 нта</a>  
21  
22 <!-- Також оновлено посилання на дії над студентом та лінки Імени  
23     та Прізвища студента -->  
24     <tr>  
25         <td>1</td>  
26         <td>  
27 ight="30" width="30" /></td>  
28         <td><a href="/students/1/edit">Корост</td>  
29         <td><a href="/students/1/edit">Андрій</td>  
30         <td>2123</td>  
31         <td>  
32             <div class="btn-group">  
33                 <button type="button" class="btn btn-default dropd\>  
34 own-toggle"
```

```
35          data-toggle="dropdown">Дія
36          <span class="caret"></span>
37      </button>
38      <ul class="dropdown-menu" role="menu">
39          <li><a href="/students/1/edit">Редагувати</a></li>
40      i>
41          <li><a href="/journal/1">Відвідування</a></li>
42          <li><a href="/students/1/delete">Видалити</a></li>
43      i>
44          </ul>
45      </div>
46  </td>
47 </tr>
```

Аналогічно можете також оновити рядочки таблиці наступних студентів.

Заготовки для в'юшок

Тепер підготуємо заготовочки для даної структури веб-адрес. А саме: наб'ємо views.py модуль в нашій аплікації демо в'юшками та заповнимо модуль urls.py в нашому проекті списком необхідних URL шаблонів.

Спочатку додамо демо в'юшку для закладки Групи та назовемо її groups_list. Додамо цю функцію у наш модуль views.py всередині аплікації:

Демо в'юшка для закладки Групи

```
1 def groups_list(request):
2     return HttpResponse('<h1>Groups Listing</h1>')
```

Тепер залінкуємо її в URL диспетчері таким чином повідомивши Django про новостворений обробник адреси /groups:

Додаємо патерн для сторінки списку груп в urls.py

```
1 from django.conf.urls import patterns, include, url
2 from django.contrib import admin
3
4 urlpatterns = patterns('',
5     # Students urls
6     url(r'^$', 'students.views.students_list', name='home'),
7     # url(r'^blog/', include('blog.urls')),
8
9     # Groups urls
10    url(r'^groups/$', 'students.views.groups_list', name='groups'),
11
12    url(r'^admin/', include(admin.site.urls)),
13
14 )
```

Як бачите, в рядку 10-му ми додали новий URL шаблон для адреси “groups/” і залінкували її на нашу нову в’юшку “groups_list”. Також дали їй назву ‘groups’, якою пізніше скористаємося в шаблоні для переходу на динамічні лінки.

Тепер, коли ми клікнемо по закладці Групи у браузері, то отримаємо сторінку із заголовком “Groups Listing”. Маємо заготовку для списку Груп, яку наб’ємо HTML кодом у подальших секціях.

Подібним чином наб’ємо демо в’юшки для форм редагування, додавання та видалення студента. А також адреси для закладки Групи:

Демо в'юшки, повний код модуля views.py

```
1 from django.shortcuts import render
2 from django.http import HttpResponseRedirect
3
4 # Views for Students
5
6 def students_list(request):
7     return render(request, 'students/students_list.html', {})
8
9 def students_add(request):
10    return HttpResponseRedirect('<h1>Student Add Form</h1>')
11
12 def students_edit(request, sid):
13    return HttpResponseRedirect('<h1>Edit Student %s</h1>' % sid)
14
15 def students_delete(request, sid):
16    return HttpResponseRedirect('<h1>Delete Student %s</h1>' % sid)
17
18 # Views for Groups
19
20 def groups_list(request):
21    return HttpResponseRedirect('<h1>Groups Listing</h1>')
22
23 def groups_add(request):
24    return HttpResponseRedirect('<h1>Group Add Form</h1>')
25
26 def groups_edit(request, gid):
27    return HttpResponseRedirect('<h1>Edit Group %s</h1>' % gid)
28
29 def groups_delete(request, gid):
30    return HttpResponseRedirect('<h1>Delete Group %s</h1>' % gid)
```

Більшість коду є подібним до нашої першої в'юшки students_list, але зверніть увагу на кілька відмінних рядків:

- 12ий: `students_edit` в'юшка приймає ще один додатковий параметр - `sid`; це є унікальний ідентифікатор студента, який передається з веб адреси сторінки на наш URL патерн, який відловлює його для нас і передає як аргумент у функцію в'юшки; далі ми побачимо як саме URL шаблон відловлює даний аргумент в адресі запиту;
- 13ий: з допомогою [інтерполяції стрічок](#)¹²⁶ в мові Python ми віддаємо у браузер заголовок сторінки, яка містить `id` студента.

Тепер повернемось до нашого модуля `urls.py` в корені проекту і додамо туди URL шаблон для кожної із нових демо в'юшок:

URL структура студентів та груп в URL шаблонах, urls.py

```
1 from django.conf.urls import patterns, include, url
2 from django.contrib import admin
3
4 urlpatterns = patterns('',
5     # Students urls
6     url(r'^$', 'students.views.students_list', name='home'),
7     url(r'^students/add/$', 'students.views.students_add',
8         name='students_add'),
9     url(r'^students/(?P<sid>\d+)/edit/$',
10         'students.views.students_edit',
11         name='students_edit'),
12     url(r'^students/(?P<sid>\d+)/delete/$',
13         'students.views.students_delete',
14         name='students_delete'),
15
16     # Groups urls
17     url(r'^groups/$', 'students.views.groups_list', name='groups'),
18     url(r'^groups/add/$', 'students.views.groups_add',
19         name='groups_add'),
20     url(r'^groups/(?P<gid>\d+)/edit/$',
21         'students.views.groups_edit',
```

¹²⁶http://www.internet-technologies.ru/articles/article_220.html

```
22         name='groups_edit'),
23     url(r'^groups/(?P<gid>\d+)/delete/$',
24         'students.views.groups_delete',
25         name='groups_delete'),
26
27     url(r'^admin/', include(admin.site.urls)),
28 )
```

Кожен із шаблонів має регулярний вираз, посилання на відповідну в'юшку та назву.

Єдина нова річ у даному коді - це регулярний вираз із групою всередині: “`^students/(?P<sid>\d+)/edit$`”. У ньому частина в круглих дужках визначає динамічну складову URL адреси - унікальний ідентифікатор студента, який може складатись із одної і більше цифр (`\d+`). Адреса типу “`students/125/edit/`” задовільнить даний регулярний вираз. А стрічка всередині кутових дужок “

Тепер рестартність свій Django сервер і перезавантажте сторінку “`http://localhost:8000/students/1/edit/`” у веб-браузері. Спробуйте поклікати по закладках Студенти, Групи, по діях: Редагувати, Видалити. У відповідь на ваші дії ви отримуватимете робочі сторінки, які є поки лише однорядковими заголовками тексту. Також зауважте як сторінка редагування студента отримує та показує ID студента переданого їй в URL адресі:



Демо сторінка редагування студента

Мова шаблонів Django

Ми уже скопіювали `students_list.html` у Django, підв'язали його до в'юшки, але ще не скористались жодними динамічними можливостями, що нам дав фреймворк в результаті даного копіювання.

Щоб мати можливість вставляти динамічні дані в HTML документі (зазвичай передані із коду в'юшки), Django фреймворк дає нам три основні інструменти:

- змінні;
- теги;
- фільтри.

Змінні призначені для вставки даних у шаблон. Наприклад, щоб вставити доступну для шаблона змінну `student_name`, потрібно скористатись наступним виразом:

```
1 {{ student_name }}
```

Даний вираз буде повністю замінений стрічкою - іменем студента. І ми можемо не лише вставляти доступну в шаблоні змінну як вона є, але й отримувати із неї атрибути, ключі та інші елементи. Уявимо, що у нас в шаблоні є доступний об'єкт `student`:

```
1 {{ student.first_name }}
```

Таким чином ми зможемо отримати доступ до атрибута студента під іменем `first_name`. При чому символ крапки (.) в змінній шаблона працює наступним чином:

- пробує отримати значення по ключу `first_name`, як із словника;
- якщо не знаходить ключа, тоді пробує отримати атрибут або метод `first_name` із об'єкта студента;
- якщо не знаходить атрибута чи метода, тоді пробує індекс списку.

Також можемо використовувати символ крапки неодноразово в одній змінній:

```
1 {{ student.group.leader }}
```

У вище наведеному виразі ми із об'єкта студента отримали групу, а із групи - старосту групи.

Фільтри призначені для модифікації значення змінної на льоту. Наприклад, для того, щоб вставити ім'я студента із символами в нижньому регістрі, можемо скористатись наступним виразом змінної:

```
1 {{ student.first_name | lower }}
```

Як бачите, фільтр - це частина виразу змінної, який слідує вкінці назви змінної після символу “|”. Можемо мати більше, ніж один фільтр в одній змінній:

```
1 {{ student.first_name | lower | safe }}
```

Також фільтри можуть приймати параметри:

```
1 {{ student.biography | truncatewords:30 }}
```

У виразі вище ми вказали вивести біографію студента з обмеженням в максимум 30 слів. Як бачите параметр фільтра ми передали йому через символ двокрапки (:).

Найпоширенішими фільтрами є:

- *length*: повертає довжину змінної;
- *default*: повертає значення по-замовчуванню, якщо значення змінної порожнє або False;
- *safe*: забороняє конвертувати зарезервовані HTML символи у HTML Entities;
- *striptags*: видаляє усі теги з тексту;
- *urlencode*: адаптує символи стрічкі у потрібний для URL адреси формат;
- *upper* i *lower*: конвертує текст у верхній та нижній регістр відповідно.

Теги в Django шаблонах є найскладнішою динамічною компонентою. Вони використовуються для вставки даних, для реалізації логіки та потоку коду в шаблоні, а також для завантаження додаткової інформації в шаблон.

Тег в Django шаблоні виглядає наступним чином:

```
1 {% tag %}
```

Бувають теги як одинарні, так і парні (тобто із закриваючим тегом). Парні теги виглядають наступним чином:

```
1  {% if student %}  
2      {{ student.first_name }}  
3  {% endif %}
```

Тобто закриваючий тег завжди маю вигляд “`{% endtag %}`”, де “tag” - це назва відкриваючого тега. В прикладі вище ми використали умовний тег “`{% if %}`”. Він дозволяє лише при певній умові вставляти той чи інший контент в сторінці.

Другим надзвичайно поширеним тегом є тег циклу. Ось приклад його використання:

Ось як можна динамічно генерувати список в HTML документі маючи тег циклу `for`

```
1  <ul>  
2  {% for student in students %}  
3      <li>{{ student.first_name }}</li>  
4  {% endfor %}  
5  </ul>
```

Крім того теги можна використовувати вкладаючи один в інший:

```
1  <ul>  
2  {% for student in students %}  
3      {% if student.age > 22 %}  
4          <li>{{ student.first_name }}</li>  
5      {% endif %}  
6  {% endfor %}  
7  </ul>
```

Для коментарів всередині Django шаблоні можна скористатись спеціальним тегом коментаря:

```
1  {% comment "Текст коментаря" %}  
2      HTML код або Django теги  
3  {% endcomment %}
```

Також широко використовуються теги “block” та “extends”, які дозволяють реалізовувати унаслідування в Django шаблонах. Але про це ми поговоримо детальніше у наступних секціях даної глави.

Загалом в Django є близько 20 вбудованих тегів. [Тут¹²⁷](#) і [тут¹²⁸](#) ви зможете детально ознайомитися із усіма можливостями Django шаблонів.

Тепер, поверхнево ознайомившись із основами мови Django шаблонів, будемо рухатись до наступної секції. Усі нові теги, фільтри та змінні ми будемо окремо деталізувати і вивчати в процесі роботи над проектом.

Динамізуємо URL адреси

А тепер уявіть, що ми хочемо змінити лінк на закладці Групи із /groups на /mygroups. Для цього в даний момент треба буде оновити код як мінімум у двох місцях: urls.py - модуль URL шаблонів та students_list.html - шаблон списку студентів. А пізніше, коли ми розробимо ще з десяток шаблонів, така зміна призведе до годинної роботи програміста.

Саме тому, щоб уникати подібних незручностей варто користуватись динамічними URL адресами у Django шаблонах. Таким чином, щоб єдине місце де вказується формат URL адреси був модуль urls.py, а всі інші місця в проекті, що мають справу із URL адресами, брали інформацію із модуля urls.py.

У цьому нам допоможуть Django шаблони та тег “url”. Він є ще одним поширенім тегом, який ми інтенсивно використовуватимемо у даній секції. Цей тег повертає згенерований URL шлях (PATH - URL адреса без частини з доменом), що відповідає певній зареєстрованій в’ющі або назві URL шаблона (аргумент name у виклику функції url всередині модуля urls.py). Ми в основному використовуватимемо іменовані URL патерни в urls.py і, відповідно, передавати тегу “url” саме назву цих патернів.

¹²⁷<http://djbook.ru/rel1.7/topics/templates.html>

¹²⁸<http://djbook.ru/rel1.7/ref/templates/builtins.html#ref-templates-builtins-tags>

Перейдемо одразу до прикладу і побачимо як можна отримати URL шлях до сторінки додавання студента:

URL шлях до сторінки із формою додавання нового студента

```
1  {% url "students_add" %}
```

Тегу “url” ми передали параметром назву (name=“students_add” в urls.py модулі) URL шаблона, який відповідає за в’юшку додавання нового студента. І тепер, якщо ми замінимо даним виразом значення атрибуту href всередині тегу лінка Додати Студента, ми отримаємо то й же ж результат, що і маємо зараз: “/students/add/” шлях. Але у даному випадку ми маємо дуже велику перевагу. Якщо ми захочемо змінити адресу, по якій буде доступна наша форма додавання, нам треба буде просто оновити URL шаблон в urls.py, а лінк в шаблоні буде автоматично вказувати на нову адресу.

Спробуйте і поексперимейтуйте із зміною URL шаблона для нашої форми додавання студента і подивіться чи дійсно все працює як описано в попередньому параграфі.

Крім того тег “url” може приймати додаткові параметри, що є групами в URL шаблонах. Наприклад, наша URL адреса форми редагування студента містить групу “sid”, яка є унікальним ідентифікатором студента в системі. Таким чином, щоб отримати URL шлях до такої форми, використовуючи “url” тег, нам потрібно також додатково передавати ID студента:

Передаємо ID студента тегу url

```
1  {% url "students_edit" 125 %}
```

Таким чином ми отримаємо URL шлях “/students/125/edit/”.

Тепер, ознайомившись і зрозумівши як працює даний тег, а особливо знаючи його користь для нас, беремось за наш шаблон students_list.html і замінюємо усі ті адреси лінків, в яких ми поки вручну прописали шляхи:

- закладки навігації;
- форми редагування студента і групи;
- форми додавання студента і групи;
- форми видалення студента і групи.

Увага. Ми можемо замінити лише ті посилання, для яких уже існують заготовлені в'юшки та URL шаблони. Ми не можемо поки оновити посилання для неіснуючих поки в'юшок.

Таким чином оновлений HTML файл матиме наступні зміни:

Оновлені href атрибути лінків з тегом url. Наведено файл лише частинами.

```
1 <!-- Навігація на сторінці з оновленими атрибутами href.
2   Відвідування ми поки не чіпали, адже немає під неї в'юшки. -->
3   <!-- Start subheader -->
4   <div class="row" id="sub-header">
5     <div class="col-xs-12">
6       <ul class="nav nav-tabs" role="tablist">
7         <li role="presentation" class="active"><a href="{% url
8           "home" %}">Студенти</a></li>
9         <li role="presentation"><a href="/journal">Відвідування
10          </a></li>
11         <li role="presentation"><a href="{% url "groups" %}">Групи
12          </a></li>
13       </ul>
14     </div>
15   </div>
16   <!-- End subheader -->
17
18 <!-- Оновлена кнопка Додати Студента -->
19   <h2>База Студентів</h2>
20   <a href="{% url "students_add" %}" class="btn btn-primary">
```

```
21             Додати Студента</a>
22
23 <!-- Оновлений перший рядок таблиці із студентом -->
24     <tr>
25         <td>1</td>
26         <td></td>
28         <td><a href="{% url "students_edit" 1 %}">Корост</td>
29         <td><a href="{% url "students_edit" 1 %}">Андрій</td>
30         <td>2123</td>
31         <td>
32             <div class="btn-group">
33                 <button type="button" class="btn btn-default
34                     dropdown-toggle" data-toggle="dropdown">Дія
35                 <span class="caret"></span>
36             </button>
37             <ul class="dropdown-menu" role="menu">
38                 <li><a href="{% url "students_edit" 1 %}">
39                     Редагувати</a></li>
40                 <li><a href="/journal/1">Відвідування</a></li>
41                 <li><a href="{% url "students_delete" 1 %}">
42                     Видалити</a></li>
43             </ul>
44         </div>
45     </td>
46 </tr>
```

Думаю у вище наведеному коді для вас уже немає бути нічого нового. Ми просто замінили частину href атрибутів із напряму вказаних шляхів на використання тегів “url” для існуючих Django в'юшок.

Усі лінки пов’язані із журналом відвідування ми опрацюємо в одній із наступних глав, коли розбиратимемо технологію Ajax в контексті Django.

Перед тим як переходити до наступної секції закомітьте і запуште усі останні зміни в репозиторій коду. Самостійно. І намагайтесь після кожної логічної групи змін робити окремий коміт. Це покращує історію змін в репозиторії та, при потребі, полегшує повернення до попередніх версій коду.

Статичні ресурси

Ви ще не забули, що наша сторінка із списком студентів виглядає поки не дуже? Адже ми ще й досі не пофіксили статичні ресурси, що мають буде під'єднані у Django.

Маємо як мінімум дві задачі:

1. Зробити так, щоб наш main.css файл запрацював на нашій сторінці.
2. Переконатись, що зображення студентів відображаються коректно в таблиці.

До статичних ресурсів на веб-сторінці зазвичай належать CSS, Javascript файли, а також зображення. Кожен із даних ресурсів підтягується браузером, в окремому запиті після отримання головної HTML сторінки. Згідно останніх практик усі ці типи ресурсів краще тримати на зовнішніх CDN сервісах, або як мінімум обслуговувати поза Django URL диспетчатор (тобто напряму через фронт-енд сервер, напр. Nginx або Apache).

static

Слово “static” використовується у двох місцях при роботі із статичними ресурсами в Django:

- по-замовчуванню фреймворк заглядає у кожну заінсталювану аплікацію і перевіряє чи часом там немає підпапки “static”; якщо вона є, тоді Django автоматично додасть її у свій реєстр статичних папок і відповідно кожен внутрішній файл і папка будуть доступні через браузер під певною адресою;
- по-замовчуванню змінна STATIC_URL в модулі settings.py проекта є рівна стрічці “/static/”; це означає, що статичні ресурси у веб-переглядачі обслуговуються в під-секції URL адрес “/static/”.

Таким чином нашим першим кроком має бути перенесення усіх наших статичних ресурсів у підпапку static всередині аплікації “students”:

Переносимо статичні ресурси в папку static всередині аплікації

```
1 # заходимо в корінь нашої аплікації students
2 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students
3
4 # створюємо підпапку static
5 $ mkdir static
6
7 # рекурсивно копіюємо сюди ресурси із папки static, що лежать в
8 # корені репозиторію
9 cp -R ../static/* static/
```

Тепер, якщо ви введете в рядочку адреса вашого браузера наступну адресу: “<http://localhost:8000/static/css/main.css>”, отримаєте у відповідь текст файлу main.css. Це буде підтвердженням того, що ви все зробили правильно.

Завдяки прописаній змінній в settings.py в корені проекту:

```
1 STATIC_URL = '/static/'
```

а також дефолтному списку так званих завантажувачів (loaders) статичних ресурсів із аплікацій, ресурси всередині нашої папки static автоматично стали доступними у веб-переглядачі під шляхом ‘/static/’.

Більше про те, як працює підключення статичних ресурсів в Django можете прочитати [тут¹²⁹](#).

Але спробуйте тепер перевантажити кореневу сторінку вашої аплікації...

Ніяких змін? Далі поламані стилі меню груп? Це і не дивно, адже ми ще не оновили посилання на main.css файл всередині students_list.html шаблону. Тепер ми знаємо, під якою веб-адресою є доступні наші статичні ресурси і тому з легкістю можемо пофіксити дану проблему:

Оновлений тег link, що під'єднує файл стилів main.css

```
1 <link rel="stylesheet"
2   href="/static/css/main.css">
```

Все доволі просто.

Подібну річ можемо проробити із тегами зображень студентів. Адже ми також скопіювали уже підпапку "img" в статичну директорію. Відповідно кожне із зображень є доступним під наступним посиланням: "/static/img/<filename>".

Спробуйте оновити дані теги зображень самостійно і заставити зображення коректно спрацювати в списку студентів.

Насправді папка static слугує для зберігання статичних ресурсів, що належать до функціоналу самої аплікації. Будь-який контент, що є даними аплікації, а не частиною функціоналу, має йти або в базу даних, або у папку media. Проте, допоки ми не реалізували моделей для студентів, ми використовуємо для них статичні зображення замість зображень розташованих у папці media. Більше про це у наступній главі.

¹²⁹<http://djbook.ru/rel1.7/ref/contrib/staticfiles.html>

В коді, що йде разом із книгою ви можете побачити кінцевий результат із усіма реалізованими завданнями та функціоналом. Проте я хочу, щоб ви заглядали туди лише після того, як спробуєте зробити домашку самі. Так ви отримаєте більше практики та розуміння фреймворку. Код, що йде разом із книгою слугує швидше інструментом для самоперевірки.

Дебаг статики

Питання підключення статичних ресурсів, зокрема файлів CSS та Javascript, є доволі проблематичним питанням для початківців. Саме тому я вирішив виділити даній темі окрему невелику секцію. У ній ми оглянемо як краще і в якій послідовності відслідковувати проблеми із непрацюючими ресурсами.

Основна складність у тому, що є як мінімум 5 різних рівнів, де справи можуть піти не так із підключенням ресурсу на сторінці:

- HTML документ;
- налаштування проекту;
- URL диспетчер Django;
- статична папка в аплікації;
- побудова URL адреси в HTML документі.

При тестуванні та пошуку причини, чому ваш статичний CSS чи Javascript файл не працює, користуйтесь Firebug плагіном у Firefox. Він дасть вам перший крок до можливої проблеми. Закладка Net надзвичайно корисний інструмент відслідковування усіх ресурсів, які браузер збирає для HTML сторінки відповідно до її тегів.

Отже, покрокова інструкція. Важливо притримуватись її у цьому порядку:

1. перевірте чи тег, що підключає ваш CSS чи Javascript файл на сторінку взагалі присутній в HTML коді сторінки;
2. якщо присутній, перевірте чи він має усі необхідні атрибути (напр. link повинен мати атрибут rel= "stylesheet" для підключення CSS файлу);
3. далі перевірте чи правильне посилання в тезі, що підключає статичний ресурс;
4. якщо правильне, тоді вводите в браузері (в новій закладці) дане посилання і перевіряете чи браузер повертає текст цього файлу;
5. якщо видає 404 код - Not Found сторінку, перевіряєте чи settings.py містить правильне значення у змінній STATIC_URL; має співпадати із першою компонентою вашої URL адреси до статичного ресурсу (напр. для /static/my.css - це частина /static/);
6. якщо все добре, тоді перевіряєте чи ви створили "static" папку в правильному місці: якщо всередині заіnstальованої аплікації, тоді нічого робити додатково не потрібно; якщо ж в інших місцях, тоді потрібно окремо додавати шлях до неї в змінній STATICFILES_DIRS¹³⁰ всередині модуля settings.py вашого Django проекту;
7. якщо все гаразд і папка підключена у Django, тоді переконайтесь, що ваш статичний файл існує в тій статичній папці і знаходиться у правильному місці і з коректною назвою (тією, що згадується в URL адресі даного ресурсу);
8. якщо все і тут добре, але далі ваш ресурс не підключається на сторінку, тоді телефонуйте 911, або хоча б мені (+38 (097) 998-73-48) - будемо екстренно розбиратись! ;)

Думаю вище наведений алгоритм допоможе зекономити масу постів із запитаннями у закритій групі підтримці. Але якщо ви зможете знайти ще інші причини чому статичний ресурс не спрацював - діліться, будемо покращувати дану інструкцію.

¹³⁰http://djbook.ru/rel1.7/ref/settings.html#std:setting-STATICFILES_DIRS

Змінна STATICFILES_DIRS в settings.py починаючи з Django 1.7 версії по замовчуванню не використовується в шаблоні проекту. Дефолтний список завантажувачів статичних ресурсів включає у себе "django.contrib.staticfiles.finders.AppDirectoriesFinder", який автоматично знаходить нашу папку "static" всередині нашої аплікації. Якщо в майбутньому нам потрібно буде підключити статичну папку, що не є всередині аплікації, тоді її треба буде додати до списку STATICFILES_DIRS всередині модуля settings.py.

Тег static

На закінчення секції про статичні ресурси, давайте ознайомимось із ще одним тегом в Django шаблонах. Він зможе замінити нам частину URL адреси в шляхах до статичних ресурсів. А саме, STATIC_URL стрічку, яка вказана у settings.py модулі проекту.

Таким чином, переключившись на використання тегу static в наших шаблонах, ми зможемо оновити змінну STATIC_URL наприклад з /static/ на /myresources/ і далі мати абсолютно працючу сторінку без жодних додаткових змін.

Отже, яким чином ми заміняємо наші посилання на статичні ресурси на сторінці?

Приклад використання тегу static для посилань на статичні ресурси

```
1  {% load static from staticfiles %}  
2  
3  <!DOCTYPE html>  
4  <html lang="uk">  
5    <head>  
6      <meta charset="UTF-8"/>  
7      <title>Сервіс Обліку Студентів</title>  
8      <meta name="description"  
9        value="Система Обліку Студентів Навчального Закладу" />
```

```
10 <meta name="viewport"  
11     content="width=device-width, initial-scale=1">  
12  
13 <link rel="stylesheet"  
14     href="https://cdn.jsdelivr.net/bootstrap/3.3.0/css/bootstrap\  
15 .min.css">  
16 <link rel="stylesheet"  
17     href="{% static "css/main.css" %}">  
18 </head>
```

Ми спеціально навели усю шапку нашого HTML документа, щоб показати, що на початку файла ми додали новий тег “load”, який імпортує тег “static” у наш шаблон. Тег “static” не є по замовчуванню доступний в Django шаблонах, тому перед його використанням ми маємо завжди імпортувати його із аплікації “staticfiles”.

Таким самим чином ми можемо імпортувати будь-які інші, в тому числі наші власні, додаткові теги в шаблон.

На передостанньому рядку в прикладі вище ми скористалися тегом “static”, якому передали шлях до main.css файлу всередині папки static. Таким чином тег “static” поверне нам у шаблон стрічку: /static/css/main.css. Після такої зміни посилання на даний ресурс завжди буде актуальним навіть після оновлення STATIC_URL змінної всередині модуля settings.py.

На домашнє завдання: оновіть таким самим чином теги зображень студентів, щоб вони не лише працювали, але й працювали через тег “static”.

Динамізуємо список студентів

Щоб у повну силу попрактикуватись із мовою Django шаблонів, в якості останнього завдання даної секції, динамізуємо список студентів.

Шаблон отримуватиме змінну “students” із нашої в’юшки. Данна змінна буде списком студентів, а шаблон пробігатиметься по даному списку і формуватиме рядочки таблиці із усіма необхідними вставками даних із переданого їй списку.

Звісно, поки що ми не маємо моделей і даних наших студентів. Тому, на даний момент, ми вручну сформуємо “прибитий” список студентів, в якому кожен елемент (студент) буде словником із ключами та значеннями (даними про студента).

Ось як виглядатиме оновлений код в’юшки students_list:

Передаємо у шаблон вручну заготований список даних про студентів

```
1 # -*- coding: utf-8 -*-
2
3 from django.shortcuts import render
4 from django.http import HttpResponseRedirect
5
6 # Views for Students
7
8 def students_list(request):
9     students = (
10         {'id': 1,
11          'first_name': u'Віталій',
12          'last_name': u'Подоба',
13          'ticket': 235,
14          'image': 'img/me.jpeg'},
15         {'id': 2,
16          'first_name': u'Корост',
17          'last_name': u'Андрій',
18          'ticket': 2123,
19          'image': 'img/piv.png'},
```

```
20      )
21      return render(request, 'students/students_list.html',
22      {'students': students})
```

Зверніть увагу на наступні рядочки в оновленій функції:

- 1ий: у файлі ми використали не ASCII символи, тому треба було додати декларацію про кодування UTF-8; в протилежному випадку Python інтерпретатор ламається;
- 9ий: декларуємо змінну `students` і присвоюємо їй кортеж із двох елементів;
- 10ий: кожен із елементів кортежа є словником із ключами: `id` (унікальний ідентифікатор студента в базі), `first_name` (ім'я студента, юнікодова стрічка), `last_name` (прізвище), `ticket` (номер білета), `image` (шлях до зображення в папочці `static`);
- 22ий: третім аргументом у функцію `render` ми передаємо цього разу не порожній словник, а словник із ключем '`students`', що містить значення - список наших приготованих студентів; в шаблоні під даним ключом можна буде глобально доступатись до даного списку; цей словник - це так званий контекст шаблона, через який передаються дані із в'юшки в шаблон.

На домашнє завдання: додайте третій словник у список студентів, щоб відтворити наш оригінальний статичний список студентів.

В наступній главі ми оновимо код цієї в'юшки і отримуватимемо об'єкти студентів уже із бази даних.

Тепер, маючи доступ до списку заготованих студентів можемо переходити до оновлення нашого шаблону. Для цього скористаємось тегом “{% for %}”, щоб пробігтись по усіх студентах, в середині якого вставляти мемо рядок таблиці (тег tr).

Крім того, усі імена студентів, номер білету, порядковий номер студента, а також лінки на Дії над студентом, ми динамічно вставляти мемо використовуючи дані із списку студентів та Django змінні в шаблонах. Маючи такий тег циклу, ми можемо видаляти решту статичних рядочків таблиці, залишивши лише перший, який і огорнемо в тег циклу. По-суті, це як у мові програмування Python з використанням циклу “for”. Ось оновлене тіло таблиці студентів:

Заповнюємо таблицю із студентами динамічно із змінної students

```
1      <tbody>
2          {% for student in students %}
3              <tr>
4                  <td>{{ forloop.counter }}</td>
5                  <td></td>
8                  <td><a href="{% url "students_edit" student.id %}">
9                      {{ student.last_name }}</td>
10                 <td><a href="{% url "students_edit" student.id %}">
11                     {{ student.first_name }}</td>
12                 <td>{{ student.ticket }}</td>
13                 <td>
14                     <div class="btn-group">
15                         <button type="button"
16                             class="btn btn-default dropdown-toggle"
17                             data-toggle="dropdown">Дія
18                             <span class="caret"></span>
19                         </button>
20                         <ul class="dropdown-menu" role="menu">
21                             <li><a
22                                 href="{% url "students_edit" student.id %}">
```

```
23          Редагувати</a></li>
24      <li><a
25          href="/journal/{{ student.id }}">
26              Відвідування</a></li>
27      <li><a
28          href="{% url "students_delete" student.id %}">
29              Видалити</a></li>
30      </ul>
31  </div>
32 </td>
33 </tr>
34 {% endfor %}
35
36 </tbody>
```

Давайте проаналізуємо найважливіші моменти даного оновлення таблиці із студентами:

- 2ий рядок: в циклі for пробігаємось по студентах, змінній student при-
своюємо значення поточного студента в циклі (словника);
- 3ий: одразу після тегу циклу йде тег рядка таблиці (tr); це означає, що
цикл for повторюватиме у вихідному HTML документі рядки таблиці
рівно стільки разів, скільки є елементів у списку students;
- 4ий: всередині циклу Django шаблон дає нам доступ до змінної forloop,
яка в свою чергу надає інформацію про поточну ітерацію в циклі; напри-
клад атрибут forloop.counter дає нам порядковий номер ітерації циклу
починаючи з 1; таким чином ми класно використали його для генерації
порядкового номера студента у першій колонці таблиці;
- 5ий: “{% static student.image %}” - як бачите, всередині тега змінну не
потрібно огортати додатково у подвійні фігурні дужки; і тим більше
не можна огортати у подвійні лапки, інакше шаблон зрозуміє стрічку
“student.image” дослівно як статичну стрічку, а не змінну, з якої треба
отримати атрибут “image”;
- 8ий: подібним чином ми передаємо аргумент “student.id” url тегу для
генерації URL адреси сторінки редагування поточного в циклі студента;

- 9ий: “{{ student.last_name }}” - просто вставляємо прізвище студента дістаючи його із словника “student” по ключу “last_name”.

Усі інші рядки даного шаблону є подібними і, думаю, не потребують додаткового пояснення. Після застосування цих змін спробуйте оновити вашу сторінку. Якщо нічого не змінилось, немає помилок і ви далі бачите той самий список студентів, тоді все працює!

Тепер, маючи студентів в базі даних, буде дуже просто підключити їх, оновивши код в'юшки і практично не чіпаючи код шаблона. Класно?

Знову ж таки, не забудьте перед переходом до наступної секції закинути усі останні зміни в репозиторій. Користуйтесь командами “git status” та “git diff” перед кожним комітом.

Реалізуємо закладку Групи

У цій главі ми також реалізуємо закладку Групи, на якій пізніше буде список груп із бази даних.

Сервіс Обліку Студентів

Група: Усі Студенти

#	Назва ↑	Староста	Дії
1	МтМ-21	Ячменев Олег	Дія ▾
2	МтМ-22	Віталій Подоба	Дія ▾
3	МтМ-23	Іванов Андрій	Дія ▾

1 2 Далі

© 2013 Сервіс Бази Студентів

Список Груп

Проте не лякайтесь. Дане завдання забере у нас значно менше часу, ніж ми присвятили на початкову реалізацію сторінки із списком студентів.

Уявіть собі, що вам потрібно для того, щоб тепер створити сторінку із групами:

- скопіювати шаблон `students_list.html` у `groups_list.html`;
- передагувати даний шаблон і оновити увесь контент на сторінці, залишаючи глобальні елементи (меню груп, лого, навігація, футер) практично незмінними;
- оновити демо в'юшку `groups_listing`, щоб рендерила наш новий шаблон для груп.

Ніби процес логічний, але є одне але. Пізніше ми створимо шаблони для журналу відвідування, форми редагування студента (групи), форм видалення та додавання студента (групи). Загалом у нас буде близько 15 шаблонів. І уявіть, що відбудеться, якщо нам потрібно буде додати в головну навігацію нову закладку: Статистика.

Для цього най прийдеться пройтись по усіх цих шаблонах і додати у кожен із них нову закладку навігації. Я думаю підтримка такоїapplікації буде більше займати часу, ніж її початкова реалізація.

Унаслідування

Саме для того, щоб уникати подібних проблем і винайшли так звані макроси, блоки, концепції унаслідування, імпортування та будь-якого іншого перевикористання уже існуючого функціоналу в різних місцях проекту.

В Python маємо ООП та поняття унаслідування класів. Це дозволяє нам мати базовий клас із спільним функціоналом для інших класів, які будуть його унаслідувати, щоб набути цей функціонал. Таким чином уникається дублювання коду та значно пришвидшується темп його розробки.

В Django шаблонах є подібна концепція унаслідування, яка дозволяє мати один базовий шаблон із спільними для більшості шаблонів елементами, а унікальні для кожної окремої сторінки елементи мають спеціально відведені

блоки в цьому базовому шаблоні, куди ці сторінки можуть вносити власні зміни.

Таким чином в Django маємо два важливих теги, що дозволять нам скористатись унаслідуванням та уникнути дубляжу повторюваних елементів на сторінках нашого проекту:

- *block*: в базовому шаблоні він декларує блок, який може перебити шаблон, що його унаслідує; в кінцевому шаблоні цей тег перебиває власним контентом контент блока із такою самою назвою в базовому шаблоні;
- *extends*: використовується в кінцевому шаблоні, щоб вказати з якого шаблону даний шаблон унаслідується.

Давайте на прикладі розробки базового шаблону для нашої аплікації розглянемо близче використання даних тегів:

Оформляємо base.html

В першу чергу визначимось, які саме секції будуть повторюватись практично без змін на кожній із наших сторінок:

- логотип;
- меню груп;
- навігаційні закладки: Студенти, Відвідування та Групи;
- нижній колонитул.

Крім того, мабуть кожна сторінка матиме заголовок рівня h2, де на головній сторінці у нас є “База Студентів”. Просто на кожній сторінці ми матимемо інший текст всередині тегу h2. Таким чином для вмісту h2 елемента ми також можемо виділити окремий блок, щоб потім користуватись ним в кінцевих шаблонах.

Отже, копіюємо файл students_list.html в base.html:

```
1 # заходимо в папку із нашими шаблонами
2 $ cd students/templates/students
3
4 # копіюємо students_list.html в base.html
5 $ cp students_list.html base.html
```

Відкриваємо новостворений файл в улюбленому редакторі і залишаємо у ньому лише повторювані елементи, замінюючи усе решта іменованими блоками:

base.html

```
1 {% load static from staticfiles %} 
2 <!DOCTYPE html>
3 <html lang="uk">
4   <head>
5     <meta charset="UTF-8"/>
6     <title>Сервіс Обліку Студентів - {% block meta_title %}{% endblock \
7       meta_title %}</title>
8     <meta name="description" value="Система Обліку Студентів Навчально \
9       го Закладу" />
10    <meta name="viewport" content="width=device-width, initial-scale=1 \
11      ">
12
13    <link rel="stylesheet"
14      href="https://cdn.jsdelivr.net/bootstrap/3.3.0/css/bootstrap\ \
15 .min.css">
16    <link rel="stylesheet"
17      href="{% static "css/main.css" %}">
18
19    {% block extra_css %}{% endblock extra_css %}
20
21  </head>
22  <body>
23
24    <!-- Start Container -->
25    <div class="container">
```

```
26
27      <!-- Start Header -->
28      <div class="row" id="header">
29          <div class="col-xs-8">
30              <h1>Сервіс Обліку Студентів</h1>
31          </div>
32          <div class="col-xs-4" id="group-selector">
33              <strong>Група:</strong>
34              <select>
35                  <option value="">Усі Студенти</option>
36                  <option value="">Мтм-21, Віталій Подоба</option>
37                  <option value="">Мтм-22, Андрій Петров</option>
38              </select>
39          </div>
40      </div>
41      <!-- End Header -->
42
43      <!-- Start subheader -->
44      <div class="row" id="sub-header">
45          <div class="col-xs-12">
46              <ul class="nav nav-tabs" role="tablist">
47                  <li role="presentation" class="active"><a href="{% url "ho\
48 me" %}">Студенти</a></li>
49                  <li role="presentation"><a href="/journal">Відвідування</a\
50 ></li>
51                  <li role="presentation"><a href="{% url "groups" %}">Групи\
52 </a></li>
53              </ul>
54          </div>
55      </div>
56      <!-- End subheader -->
57
58      <!-- Start Content Columns -->
59      <div class="row" id="content-columns">
60          <div class="col-xs-12" id="content-column">
```

```
61          {% comment "Not used alert so far" %}  
62          <div class="alert alert-warning" role="alert">Status Update!\  
63      </div>  
64      {% endcomment %}  
65  
66  
67          <h2>{% block title %}{% endblock title %}</h2>  
68          {% block content %}{% endblock content %}  
69  
70      </div>  
71  </div>  
72  <!-- End Content Columns -->  
73  
74  <!-- Start Footer -->  
75  <div class="row" id="footer">  
76      <div class="col-xs-12">  
77          &copy; 2014 Сервіс Обліку Студентів  
78      </div>  
79  </div>  
80  <!-- End Footer -->  
81  
82  </div>  
83  <!-- End Container -->  
84  
85  <!-- Javascripts Inclusion -->  
86  <script src="http://ajax.googleapis.com/ajax/libs/jquery/2.1.1/jqu\ ery.min.js"></script>  
87  <script src="https://cdn.jsdelivr.net/bootstrap/3.3.0/js/bootstrap\ .min.js"></script>  
88  {% block extra_js %}{% endblock extra_js %}  
89  </body>  
90  </html>
```

Давайте порядково розберемо наш базовий шаблон:

- 1ий: load тег - базовий шаблон також містить завантаження тегу static; завантаження тегів в шаблон не унаслідується, тому даний load нам прийдеться мати в кожному шаблоні, і в базовому, і в кінцевому;
- 6ий: {`% block meta_title %`} - тут ми вперше скористались тегом block; ми дали йому ім'я meta_title і зможемо по даному імені вміст даного блока перекривати із кінцевого шаблона; тобто мета заголовок на наших сторінках буде у формі “Сервіс Обліку Студентів - База Даних”;
- 19ий: {`% block extra_css %`} - порожній блок на випадок, якщо один із кінцевих шаблонів потребуватиме використання свого власного окремого CSS файлу; корисно у тих випадках, коли дані стилі стосуються лише однією сторінки і тоді ми не вставляємо їх у базовий шаблон, щоб вони були доступні на кожній сторінці, а лише у тій сторінки, де дані стилі використовуються (така собі оптимізація швидкодії завантаження сторінки); подібним чином наприкінці документу перед закриваючим тегом body ми додали блок extra_js;
- логотип, меню груп та навігаційні закладки ми не змінювали; якщо вам треба буде на одній із кінцевих сторінок їх оновити, вам прийдеться огорнути їх в додаткові теги block;
- 62ий: {`% comment %`} - ми огорнули рядок статусу в тег коментаря таким чином прибралиши даний елемент тимчасово із сторінки; ми до нього повернемось в одній із наступних глав;
- 67ий: {`% block title %`} - порожній тег блоку всередині тегу h2, щоб кінцеві шаблони могли вставляти власні заголовки для сторінки і в той же час не змінювати рівень основного заголовка, а лише текст; аналогічно, якщо потрібно мати кращий контроль над HTML кодом заголовка, тоді можемо додатково огорнути ще одним блоком весь тег h2, а не лише його вміст;
- 68ий: {`% block content %`} - серце нашого документа, сюди увійде основний вміст сторінки; для кожного кінцевого шаблона він буде свій, унікальний.

Таким чином все, що робить тег “block” це визначає місця в базовому шаблоні, які можуть бути перевизначені кінцевим шаблоном.

Кінцеві шаблони також називають *дочірніми*. Таким чином дочірній шаблон унаслідується від *базового*, а базовий надає множину блоків для дочірнього для подальшої маніпуляції кодом в базового шаблоні.

В одному базовому шаблоні не може бути більше, ніж одного блоку з тією самою назвою. Інакше такі блоки будуть конфліктувати.

Тут¹³¹ ви можете знайти трохи більше деталей про унаслідування та блоки в Django.

Рефактор списку студентів та груп

Тепер глянемо на іншу сторону унаслідування - дочірній шаблон. Першим нашим шаблоном, що унаслідує від щойно створеного base.html, буде students_list.html.

Все, що нам потрібно зробити, це унаслідуватись від base.html, повикидатиувесь код, що вже є у base.html, і залишити лише унікальні частини, перед цим вклавши їх у потрібні блоки із base.html для перекриття.

Ось як виглядатиме оновлений students_list.html шаблон після унаслідування від base.html:

¹³¹<http://djbook.ru/rel1.4/topics/templates.html#template-inheritance>

students_list.html унаслідує повторювані елементи від base.html

```
1  {% extends "students/base.html" %}  
2  
3  {% load static from staticfiles %}  
4  
5  {% block meta_title %}Студенти{% endblock meta_title %}  
6  
7  {% block title %}База Студентів{% endblock title %}  
8  
9  {% block content %}  
10 <a href="{% url "students_add" %}" class="btn btn-primary">Додати Студента</a>  
11  
12  
13 <!-- Start Students Listing -->  
14 <table class="table">  
15   <thead>  
16     <tr>  
17       <th><a href="#">#</a></th>  
18       <th>Фото</th>  
19       <th><a href="#">Прізвище &uarr;</a></th>  
20       <th><a href="#">Ім'я</a></th>  
21       <th><a href="#">№ Білету</a></th>  
22       <th>Дії</th>  
23     </tr>  
24   </thead>  
25   <tbody>  
26     {% for student in students %}  
27       <tr>  
28         <td>{{ forloop.counter }}</td>  
29         <td></td>  
30       <td><a href="{% url "students_edit" student.id %}">{{ student.last_name }}</a>  
31       <td><a href="{% url "students_edit" student.id %}">{{ student.first_name }}</a>  
32     </tr>
```

```
35      <td>{{ student.ticket }}</td>
36      <td>
37          <div class="btn-group">
38              <button type="button" class="btn btn-default dropdown-togg\
39  le"
40                  data-toggle="dropdown">Дія
41                  <span class="caret"></span>
42              </button>
43              <ul class="dropdown-menu" role="menu">
44                  <li><a href="{% url "students_edit" student.id %}">Редаг\
45 увати</a></li>
46                  <li><a href="/journal/{{ student.id }}"/>Відвідування</a>\<
47  /li>
48                  <li><a href="{% url "students_delete" student.id %}">Вид\\
49  алити</a></li>
50          </ul>
51      </div>
52      </td>
53  </tr>
54  {% endfor %}
55
56  </tbody>
57  </table>
58  <!-- End Students Listing -->
59
60  <nav>
61      <ul class="pagination">
62          <li><a href="#">&laquo;</a></li>
63          <li><a href="#">1</a></li>
64          <li class="active"><a href="#">2</a></li>
65          <li><a href="#">3</a></li>
66          <li><a href="#">4</a></li>
67          <li><a href="#">5</a></li>
68          <li><a href="#">&raquo;</a></li>
69      </ul>
```

```
70 </nav>
71
72 {% endblock content %}
```

Можете бачити на скільки тепер спростився наш шаблон із студентами! Ми залишили в ньому лише унікальні частини саме для сторінки із списком студентів, а все решта отримали від базового шаблона. А тепер давайте оглянемо найважливіші моменти останніх оновлень:

- 1ий рядок: extends - обов'язково даний тег має йти першим в дочірньому шаблоні; він вказує інтерпретатору шаблонів, що ми унаслідуємось від шаблона “students/base.html”;
- 3ий: load: так, завантаження додаткових тегів потрібно робити і в базовому і в дочірньому тезі; даний функціонал не передається при унаслідуванні шаблонів;
- 5ий: {*block meta_title*} - додаємо суфікс “Студенти” до мета-заголовку сторінки;
- 7ий: вставляємо заголовок на сторінку “База Студентів”;
- 9ий: вставляємо весь контент сторінки без змін: кнопка Додати Студента та таблиця із студентами;
- 72ий: {*endblock content*} - не забуваємо закривати усі теги блоків.

Без тегу “block” ми не зможемо вставити жоден текст в базовий шаблон. Тому наш оновлений students_list.html шаблон містить увесь контент виключно всередині даних тегів.

Рівнів унаслідування може бути багато. Тобто від нашого students_list.html також може унаслідуватись інший шаблон і знову ж таки надписати свій контент поверх визначених блоків.

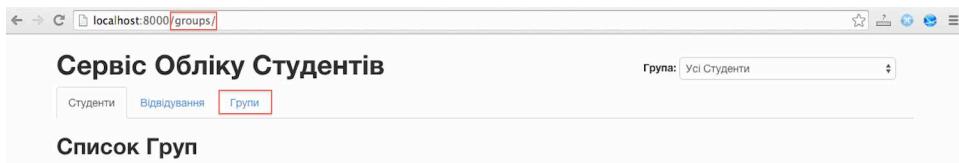
На домашнє завдання: в попередній главі у вас на домашнє завдання було зверстати сторінку із списком груп. Так от, тепер її треба оновити подібно до шаблона із списком студентів, щоб ми використали унаслідування, а також усі лінки були динамізовані з допомогою тегу “url”, а статичні ресурси коректно підключені до сторінки. Попрошу вас не заглядати у вихідний код проекту, що йде разом із даною главою книги, а лише звіритись його для самоконтролю після завершення даного завдання. Підказка: треба просто скопіювати `students_list.html` у шаблон для списку груп і трохи підправити.

Як вам тепер вигляд наших спрощених шаблонів із списком студентів та груп? Механізм унаслідування в Django шаблонах ще не один раз дозволить зберегти нам масу часу.

Знову комітимо наші останні змінні, не забуваючи при цьому додати нові файли у коміт (`git add [filename]`). Також закидуємо зміни на віддалений репозиторій.

Процесор контексту

Спробуйте тепер походити між закладками Студенти та Групи. Працює непогано, так? Але мабуть ви зауважили один нюанс: в якій секції ви б не знаходились, все одно вибраною є закладка Студенти. Це тому, що ми статично приobili HTML клас “active” елементу навігації Студенти:



Будучи на сторінці `/groups` закладки Групи не є підсвіченою як поточна

Щоб виправити дану ситуацію, ми можемо в базовому шаблоні вставити тег умови (“% if %”) для кожного із елементів в закладках, який перевірятиме поточну URL адресу запиту і відповідно до цього додаватиме клас “active” лише до однієї із закладок. Відповідно до поточної секції сайту, де ми знаходимось.

Що таке процесор?

Проте, щоб зробити це нам пригодиться інформація про поточний шлях до сторінки, на якій ми знаходимось. Цю інформацію ми можемо отримати із об'єкта запиту (request), атрибут `path`¹³².

По замовчуванню фреймворк Django не вкладає змінну `request` як доступну в середовищі шаблонів. Але є вбудований процесор контексту, який для нас це може зробити.

Процесор контексту¹³³ (context processor) - це функція, яка приймає об'єкт запиту і повертає словник із даними, що стають автоматично доступними у кожному шаблоні проекту, який використовує даний процесор. Якщо у більшість шаблонів у вашому проекті чи аплікації ви передаєте один і той самий кусок даних з ваших в'юшок, тоді варто задуматись про перенесення таких даних у ваш кастомний процесор контексту. Таким чином написавши один раз логіку по підготовці даних, ви отримаєте їх у кожному вашому шаблоні.

Усі процесори контексту реєструються в налаштуваннях проекту (`settings.py`) з допомогою змінної: `TEMPLATE_CONTEXT_PROCESSORS`. Зверніть увагу, що ми не просто встановили дану змінну у список із одного елемента із посиланням на новий процесор, а також додали усі дефолтні Django процесори. Інакше наш список процесорів надпише дефолтний список і ми залишимось із поламаним проектом через брак процесорів:

¹³²<http://djbook.ru/rel1.7/ref/request-response.html#django.http.HttpRequest.path>

¹³³<http://softwaremaniacs.org/blog/2006/01/12/context-processors/>

Додаємо вбудований Django процесор ‘request’ до списку процесорів контексту нашого проекту, settings.py

```
1 from django.conf import global_settings
2
3 TEMPLATE_CONTEXT_PROCESSORS = \
4     global_settings.TEMPLATE_CONTEXT_PROCESSORS + (
5         "django.core.context_processors.request",
6     )
```

У першому рядку ми імпортуємо дефолтні налаштування Django (global_settings модуль), які потім використовуємо, щоб набити наш список TEMPLATE_CONTEXT_PROCESSORS дефолтними процесорами. В передостанньому рядочку ми додали вбудований процесор “django.core.context_processors.request”, який і зробить для нас усю магію.

Вибрана закладка в навігації

Після даної зміни в settings.py модулі ми отримаємо змінну “request” глобально доступною у кожному із наших шаблонів. Тому саме час оновити закладки навігації із правильно встановленим класом “active”.

Закладки навігації із динамічно визначенім класом active, base.html

```
43     <!-- Start subheader -->
44     <div class="row" id="sub-header">
45         <div class="col-xs-12">
46             <ul class="nav nav-tabs" role="tablist">
47                 <li role="presentation" {% if request.path == '/' %}class=\
48 "active"{% endif %}><a href="{% url "home" %}">Студенти</a></li>
49                 <li role="presentation" {% if '/journal' in request.path %}\
50             class="active"{% endif %}><a href="/journal">Відвідування</a></li>
51                 <li role="presentation" {% if '/groups' in request.path %}\
52             class="active"{% endif %}><a href="{% url "groups" %}">Групи</a></li\
53 >
54             </ul>
```

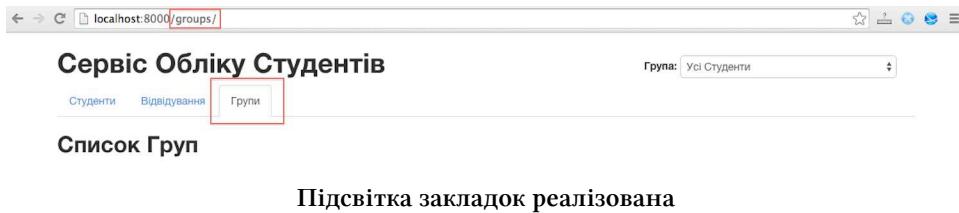
```
55      </div>
56    </div>
57  <!-- End subheader -->
```

Що ми оновили?

- 47ий: перевіряємо чи шлях поточного запиту є слешом (/), тобто ми на корені веб-сайту; якщо так, тоді вставляємо стрічку ‘class=“active”’ в тег `li` закладки Студенти;
- 51ий: перевіряємо чи поточний шлях запиту містить стрічку “/groups”; якщо так, тоді знову ж таки додаємо клас “active”; таким чином ми не лише підсвічуємо закладку Групи на сторінці із списком студентів, але й на усіх підсторінках в секціїapplікації “/groups”.

Дана реалізація не є ідеальною, адже при будь-якій зміні URL структури applікації потрібно також буде змінювати дані статичні стрічки в умовних операціях. Інакше підсвітка поточної закладки перестане працювати. Тому на домашнє завдання пропоную вам уже не таке тривіальне завдання: переробити умову всередині тегів “if” так, щоб вони використовували інформацію із urls.py URL шаблонів.

Спробуйте тепер оновити сторінку і походити між закладками Студенти та Групи. Має працювати усе чудово!



Наш власний context processor

На завершення роботи із процесорами контексту реалізуємо свій власний, який додасть нам змінну PORTAL_URL у кожен наш шаблон. Дано змінна визначатиме для нас абсолютну URL адресу кореня нашої аплікації. Вона буде корисна для формування абсолютних лінків на статичні ресурси.

Згідно кращих практик варто посилання на усі статичні ресурси формувати як абсолютні. Відносні посилання обчислюються браузером в контексті поточної сторінки таким чином не лише інколи ламаючи посилання на ресурси, але й перешкоджаючи правильному кешуванню ресурсу на стороні браузера. Адже даних два ресурси для браузера будуть виглядати як два різні файли: “/static/main.css” і “/groups/static/main.css”.

Ми уже знаємо, що процесор контексту це просто Python функція, яка приймає аргументом об'єкт запиту, а повертає словник із даними. У нашому випадку цей словник матиме єдиний ключ “PORTAL_URL” і міститиме абсолютне посилання на корінь нашої аплікації. Дане посилання ми зберігатимемо в налаштуваннях проекту (settings.py) і з легкістю зможемо його оновляти на різних серверах, зокрема на продакшині.

Додамо змінну PORTAL_URL в модуль налаштувань проекту:

Додаємо цю стрічку до settings.py

```
1 PORTAL_URL = 'http://localhost:8000'
```

У даному випадку для розробницької машини ми встановили локальну адресу Django сервера: “<http://localhost:8000>”. На продакшині це може бути щось типу “<http://www.mysite.com>”.

Другим кроком буде реалізація самого процесора. Для цього створимо новий модуль всередині нашого проекту: context_processors.py.

Створюємо модуль context_processors.py

```
1 # заходимо в корінь проекту
2 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/studentsdb
3
4 # створюємо новий файл
5 $ touch context_processors.py
```

Тепер відкриваємо новостворений модуль в редакторі і додаємо наступний код:

```
1 from .settings import PORTAL_URL
2
3 def students_proc(request):
4     return {'PORTAL_URL': PORTAL_URL}
```

Все, що ми зробили це імпортували змінну PORTAL_URL із модуля settings.py та передали її в якості ключа словника ‘PORTAL_URL’ із функції нашого кастомного процесора “students_proc”. Все досить просто та зрозуміло.

Останнім, третім кроком маємо додати нашу функцію процесора у список реєстрованих в проекті процесорів. Робимо це таким самим чином, як ми зробили при реєстрації процесора ‘request’:

Додаємо наш кастомний процесор в settings.py

```
1 TEMPLATE_CONTEXT_PROCESSORS = \
2     global_settings.TEMPLATE_CONTEXT_PROCESSORS + (
3         "django.core.context_processors.request",
4         "studentsdb.context_processors.students_proc",
5     )
```

В 4ій стрічці ми додали посилання (шлях) на нашу функцію students_proc, що знаходитьться всередині модуля context_processors. Зауважте, що в шляхам Python модулів ми не вказуємо розширень файлів “.py”.

На домашнє завдання знову не тривіальна річ: покращити наш процесор контексту, щоб він не використовував статичну змінну PORTAL_URL із модуля settings.py, а динамічно обчислював кореневу абсолютну адресу веб-сайту. Для цього скористайтеся переданим у функцію процесора об'єктом request і усіма даними, що є у ньому. Об'єднуючи різні частини даних із запиту ви зможете сформувати потрібну абсолютну адресу вашого веб-сайту.

У наступних главах ми ще повернемось до процесорів контексту, коли будемо реалізувати випадаюче меню з групами. А зараз давайте нарешті скористаємося щойно створеним процесором контексту і зробимо посилання на статичний ресурс main.css абсолютноним лінком:

Оновляємо посилання на main.css файл всередині base.html шаблону

```
1 <link rel="stylesheet"
2     href="{{ PORTAL_URL }}{% static "css/main.css" %}">
```

Ми вставили змінну “{{ PORTAL_URL }}” одразу перед тегом “static” таким чином додавши абсолютну адресу вебсайту в шляху до main.css файлу. В результаті отримаємо посилання: “<http://localhost:8000/static/css/main.css>” замість того, що ми мали “/static/css/main.css”.

Перегляньте зміни, закомітьте їх у репозиторій та закиньте коміти на віддалений сервер. Не забудьте додати новостворений файл context_processors.py.

Реорганізація в'юшок

На завершення даної глави пропоную вам самостійно реорганізувати файлик views.py в нашій аплікації у окремий підпакет views, куди поскладати окремі модулі для кожної із секцій веб-аплікації:

- groups;
- students;
- journal.

Ми уже маємо 8 функцій в'юшок в модулі `views.py`. Далі ми будемо їхню кількістю збільшувати, а логіку у кожній із функцій значно ускладнювати. Таким чином модуль `views.py` буде досить сильно розростатись в розмірі, тим самим роблячи роботу із ним складнішою. Тому надалі ми матимемо не просто окремі модулі, а окремі пакети для моделей та в'юшок.

Пам'ятайте, що при перенесенні функцій в'юшок у окремі пакети і модулі треба буде оновити відповідні посилання на них із модуля `urls.py` в нашому проекті. Також пам'ятайте, що папку можна перетворити у Python пакет з допомогою модуля `__init__.py`.

Після закінчення даного завдання зверніться у код даної глави і порівняйте із вашими результатами.

Домашнє завдання

Хух! Глава була одна із найбільших та найскладніших у даній книзі. Ми з вами пройшли масу нових для нас концепцій в Django та й вебі загалом:

- HTTP протокол;
- MVC підхід;
- Регулярні Вирази;
- Django аплікація;
- в'юшки;
- мова шаблонів;
- URL диспетчер;
- процесор контексту.

Якщо ви не все зрозуміли - не варто занадто перейматись. Ви вже молодець лише за те, що дочитали до цієї стрічки. Протягом роботи вам ще раз прийдеться повернутись до даної глави. І після деякої практики процент зрозумілого матеріалу буде все більшим і більшим.

А для того, щоб набиратись практики потрібно працювати над домашніми завданнями:

- усі ті завдання, що були згадані протягом глави у різних секціях, деякі з них ще раз повторюю нижче;
- самостійна реалізація динамічної сторінки груп із унаслідуванням, темами, лінками і статистикою;
- самостійно реорганізувати структуру в'юшок в підпапочку, не дивіться в код, що йде із книгою;
- огорніть в блоки та реалізуйте можливість змінювати меню групи, лого і футер із кінцевих шаблонів;
- переведіть вашу закладку “Відвідування”, яку ви зверстали у попередній главі на Django шаблон використовуючи усі наробки даної глави;
- спробуйте розібратись із директивою `includes` в шаблонах Django і винести `pagination` в окремий шаблон, щоб уникати дублювання даного навігаційного елементу в списку груп і в списку студентів.

Зазвітуйте ваші результати у закриту групу підтримки. Також думаю буде маса запитань - шукайте відповіді знову ж таки в групі. Без домашніх завдань користь від книги буде мінімальна!

...

В наступній главі ми з вами копнемо у “серце” будь-якої веб-аплікації: дані. Розберемось із тим, як описувати дані в Django і яке вони мають відношення безпосередньо до SQL таблиць. Оновимо списки студентів та груп уже реальними даними з бази. Навчимось користуватись міграційними командами `manage.py` скрипта і ще дещо...

Не поспішайте до наступної глави допоки достеменно не попрактикуєтесь над матеріалом поточної. Даная глава є однією із найважливіших у книзі.

7. Розробляємо моделі Студента та Групи: моделі, поля, атрибути

Ось і дійшли ми з вами до основи кожної аплікації - даних. Без даних практично неможливо реалізувати жодного проекту. В одних проектах дані створюються та управлюються працівниками, в інших - клієнтами та користувачами продукту. Але в будь-якому випадку необхідно забезпечити правильну роботу містилища даних.

В цій главі ми з вами розберемось із тим:

- що таке реляційна база даних;
- ознайомимось поверхнево із мовою SQL запитів;
- що таке ORM і з чим його їдять;
- як працювати із даними в Django;
- що таке фікстури та для чого вони потрібні взагалі.

В результаті даної глави ми:

- створимо модель студента та групи;
- оновимо списки студентів та груп, щоб працювали з базою даних;
- реалізуємо посторінкову навігацію та сортування наших лістінгів;
- промігруємо наші дані з sqlite на MySQL.

Але спочатку пройдемось трохи по теорії і розвідаємо, що таке база даних, SQL мова і взагалі ORM система.

База даних

З вікіпедії: *База даних* - впорядкований набір логічно взаємопов'язаних даних, що використовуються спільно та призначені для задоволення інформаційних потреб користувачів.

Якщо зовсім по простому, то база даних - це містилище для даних. Основним завданням бази є збереження наших даних (бажано з можливістю ефективного збереження великих обсягів) та швидка видача цих даних на наш запит.

Найбільше нас цікавитимуть саме Реляційні Бази Даних. З англійської “relation” - це відношення і означає абстрактний суто математичний термін. Проте практики трохи перекрутили даний термін і ми його зазвичай використовуємо, щоб вказати саме на таблицю. По-суті саме слово “відношення” промігувало для нас практиків у слово “таблиця”.

Таким чином реляційна база даних - це набір таблиць та зв’язків між цими таблицями. Таблиця, в свою чергу, складається із стовпців та рядків. *Стовпці* (поля) - це структура таблиці. *Рядки* - це дані таблиці.

Один рядок в таблиці представляє одну одиницю даних і містить значення (або порожнє значення) для кожного із полей таблиці. Також рядок ще називають “записом” таблиці (англ. record). Кожне поле (англ. field) таблиці має тип подібно до того, як мова програмування Python має набір своїх типів даних.

Типи даних поділяються на три категорії: числові, текстові і типи дати та часу. Детальніше із списком вбудованих типів даних бази даних MySQL можна дізнатись [тут¹³⁴](#). Найчастіше використовуваними типами в проекті будуть:

- VARCHAR;
- TEXT;
- ENUM;
- INT;
- DATE;
- DATETIME;

¹³⁴<http://bit.ly/vpmysqltypes>

- TIMESTAMP.

Тому зверніть на них особливу увагу.

Таблиця Студентів

ID	Ім'я	Прізвище	Група
1	Роман	Джексон	2
2	Джон	Ленон	1
3	Віталій	Подоба	2
4	Андрій	Іванов	4

Таблиця Груп

ID	Назва	Староста	К-сть студентів
1	МтМ-11	2	23
2	МтМ-21	1	15
3	МФК-20	3	34
4	МтМ-31	4	20

Приклад вигляду таблиць реляційної бази даних і зв'язків між ними

Таким чином поля таблиці визначають її структуру. А записи (рядки) таблиці містять самі дані.

Кожна таблиця має одне обов'язкове поле, яке унікально ідентифікує рядок в таблиці. Дане поле встановлюється автоматично в 1 для першого рядка даних в таблиці і +1 для кожного наступного (режим: autoincrement). А саме поле називають також первинним ключом (primary key). Ми будемо його часто використовувати в нашому коді.

Крім того таблиці можуть мати зв'язки між собою. А саме рядочки таблиці. В зображені вище рядочки в таблиці із студентами посилаються на рядочки із таблиці з групами. Зв'язок створюють за допомогою первинного ключа рядка. Більше про це поговоримо далі.

Нормалізація

Я лише згадаю, що дані в реляційній базі даних повинні зберігатись згідно правил “нормалізації”. Ці правила забезпечують правильний формат збереження даних та безпомилковий вивід даних із бази, а також хорошу швидкодію бази.

Нам, як початківцям, не обов’язково в даний момент глибоко вникати в дану тему. Більшість речей з нормалізації бази даних для нас робитиме обрана реалізація бази даних (а саме MySQL) в поєднанні з Django ORM системою.

Для тих, кому уже цікаво дізнатись більше про форми нормалізації даних, на [даній сторінці¹³⁵](#) можна ознайомитись із необхідним мінімумом правил.

MySQL

Як ви уже зрозуміли із глави книги про робоче середовище, в проекті ми використовуватимемо реляційну базу даних [MySQL¹³⁶](#).

Це безкоштовна реляційна база даних із відкритим кодом. Одна із найпопулярніших баз даних. Ідеально підходить для старту завдяки своїй простоті, високій швидкодії та необхідним набором стандартних властивостей.

MySQL база даних підтримує різні двигуни. Кожен із них має свої плюси та мінуси. Ми використовуватимемо саме “InnoDB”, який дає пітримку транзакцій. До MySQL 5.5.4 включно дефолтним двигателем був “MyISAM”, але починаючи з 5.5.5 версії, ним став “InnoDB”. Тому, якщо ви встановили новішу версію MySQL, нічого додатково конфігурувати не потрібно.

Якщо у вас виникають проблеми із зміною двигуна для ваших існуючих баз даних та таблиць, тоді [тут¹³⁷](#) можете дізнатись як це поправити.

¹³⁵<http://bit.ly/vpdnormalization>

¹³⁶<http://uk.wikipedia.org/wiki/MySQL>

¹³⁷<https://docs.djangoproject.com/en/1.7/ref/databases/#creating-your-tables>

В Django спільноті чомусь популярнішою є інша реляційна база з відкритим кодом - PostgreSQL. Вона має трохи більше властивостей та багатший функціонал, але і є дещо складнішою для налаштування. Тому ми користуємось саме MySQL. Перейти на PostgreSQL маючи Django не складе особливих проблем, адже Django ORM (далі ми розберемо, що таке ORM) однаково добре працює з обидвома базами даних.

SQL

Для того, щоб працювати із реляційною базою даних придумали структуровану мову запитів - [SQL¹³⁸](#) (Structured Query Language).

З її допомогою ми можемо зберігати, змінювати та отримувати дані з бази. Також можемо змінювати саму структуру бази. SQL не є повноцінною мовою програмування, а швидше інтерактивною діалоговою мовою для формування запитів в базу даних.

Великим плюсом мови SQL є те, що вона практично без особливих змін працює для різних реляційних баз даних. Більшість ваших запитів працюватимуть в MySQL, PostgreSQL і Oracle.

...

Найважливішою та найчастіше використовуваною командою SQL є [SELECT¹³⁹](#):

Робимо вибірку записів із таблиці

```
1 SELECT id, name FROM students WHERE age <= 20;
```

У команді вище з таблиці “students” ми вибираємо студентів, які мають не більше, ніж 20 років ($age \leq 20$). При чому отримуємо лише id та ім’я студентів. Даний оператор має ще багато інших додаткових груп окрім “WHERE”.

¹³⁸<http://uk.wikipedia.org/wiki/SQL>

¹³⁹[http://uk.wikipedia.org/wiki>Select_\(SQL\)](http://uk.wikipedia.org/wiki>Select_(SQL))

Далі наводжу список найважливіших SQL операторів погрупованих згідно їхнього застосування:

Зміни Структури

- **CREATE¹⁴⁰** — створення об'єкта (бази даних або таблиці);
- **ALTER¹⁴¹** — зміна об'єкта (напр. додавання чи зміна полів таблиці);
- **DROP¹⁴²** — видалення об'єкта.

Маніпуляції Даними

- **INSERT¹⁴³** — вставка запису (стрічки, рядка) в таблицю;
- **SELECT¹⁴⁴** — вибірка записів;
- **UPDATE¹⁴⁵** — зміна записів;
- **DELETE¹⁴⁶** — видалення записів із таблиці.

Права Користувача

- **GRANT¹⁴⁷** — надання прав користувачу;
- **REVOKE¹⁴⁸** — відміна заборони або дозволу користувачу.

Будь-ласка, ознайомтесь детальніше із кожним з них, а також запустіть mysql клієнт і спробуйте поекспериментувати напряму в базі даних: створіть нову базу даних, 2 таблиці: students і groups. students таблиця матиме поля: name (ім'я, тип VARCHAR), age (вік, тип INT) та group (група, посилання на рядок з таблиці groups, тип FOREIGN KEY). groups таблиця матиме поля: title (назва, тип VARCHAR), students_num (кількість студентів, тип INT). Заповніть кожну із таблиць мінімум 5-ма записами. Записи

¹⁴⁰http://www.w3schools.com/sql/sql_create_table.asp

¹⁴¹http://www.w3schools.com/sql/sql_alter.asp

¹⁴²http://www.w3schools.com/sql/sql_drop.asp

¹⁴³[http://uk.wikipedia.org/wiki/Insert_\(SQL\)](http://uk.wikipedia.org/wiki/Insert_(SQL))

¹⁴⁴[http://uk.wikipedia.org/wiki>Select_\(SQL\)](http://uk.wikipedia.org/wiki>Select_(SQL))

¹⁴⁵[http://uk.wikipedia.org/wiki/Update_\(SQL\)](http://uk.wikipedia.org/wiki/Update_(SQL))

¹⁴⁶[http://uk.wikipedia.org/wiki/Delete_\(SQL\)](http://uk.wikipedia.org/wiki/Delete_(SQL))

¹⁴⁷<http://dev.mysql.com/doc/refman/5.1/en/grant.html>

¹⁴⁸<http://dev.mysql.com/doc/refman/4.1/en/revoke.html>

студентів мають мати поле group заповнене посиланням на рядок в таблиці груп (id групи). Поекспериментуйте із вибірками даних із студентів та груп. Виберіть усіх студентів, які належать до групи, що має більше, ніж 5 студентів. Потім змініть називу поля age в таблиці студентів на student_age. І на завершення завдання: очистіть таблиці від рядків, видаліть таблиці, видаліть базу даних.

Якщо вище наведене завдання виконали, вважайте, що необхідний мінімум знань реляційної бази даних та мови SQL вже є!

Практично на кожному інтерв'ю на посаду програміста кандидата тестиують на знання баз даних та мови SQL. Окрім базових речей варто також ознайомитись із запитами, що працюють на кілька таблиць одразу - JOIN¹⁴⁹. Тут¹⁵⁰ можете ознайомитись із візуальним представленням запитів такого роду.

В проекті нам це не знадобиться, адже ми працюватимемо з базою використовуючи Django моделі і Django ORM, але попрактикуватись і розуміти, що відбувається всередині на рівні SQL мови - дуже рекомендовано! Адже не завжди буде під руками ORM система і тоді прийдеся робити запити в базу використовуючи на пряму мову SQL. Також у складніших проектах інколи приходиться писати SQL для оптимізації роботи з базою даних.

А тепер перейдемо до секції, де розберемось із класним інструментом, який дасть нам можливість не думати про мову SQL:

ORM, Django моделі

Що ж виходить? Пишемо усю логікуapplікації ми на одній мові програмування (у нашому випадку це Python), інтерфейс користувача на іншій (у нашому випадку це зв'язка HTML і CSS), а працюємо із базою даних з допомогою мови запитів SQL.

¹⁴⁹[https://ru.wikipedia.org/wiki/Join_\(SQL\)](https://ru.wikipedia.org/wiki/Join_(SQL))

¹⁵⁰<http://potapov.com.ua/library/21/>

Щоб уникати різноманіття мов необхідних для розробки проектів, програмісти постійно полегшуєть собі життя зводячи абсолютно нові технології до простіших і уже знайомих речей.

Наприклад, у веб-розробці при верстці сторінок ми користуємось по максимуму надбудовами над HTML, CSS та Javascript. Це і фреймворки із готовими веб-віджетами і бібліотеки для кросплатформенного кодування динаміки, і т.д.

ORM

Так само, щоб уникати роботи напряму із мовою SQL, а більше “сидіти” у своїй улюбленийій мові програмування, розробники створили такий собі “місток” між базою даних та об’єктами основної мови програмування. Таким чином кожного разу, коли потрібно отримати чи змінити дані в базі, програміст працює в межах своєї основної мови програмування на рівні об’єктів.

Такий місток має називу **ORM¹⁵¹** (Object Relational Mapping - Об’єктно Реляційне Відображення). Іншими словами - це співставлення записів в таблиці бази даних із концепціями об’єктно-орієнтованої мови програмування, а саме з об’єктом:

- якщо нам потрібно додати новий запис в таблиці - ми створюємо новий об’єкт з класу відповідного даній таблиці;
- якщо ми хочемо оновити існуючий рядок таблиці - через клас типу, що відповідає даній таблиці, ми отримуємо об’єкт даного рядка і змінюємо його атрибути;
- якщо ми хочемо видалити існуючий рядок таблиці - викликаємо відповідний метод на об’єкті, що відповідає даному рядку.

Вище описане співставлення - це одна із реалізацій ORM. Зокрема в Django ORM працює саме таким чином.

¹⁵¹<http://bit.ly/vrormapping>

Django моделі

Django фреймворк має свою власну ORM-ку. Вона вважається однією із найпотужніших в світі Python з точки зору функціоналу та простоти роботи із нею.

Іншою ORM системою є SQLAlchemy¹⁵², яка є незалежна від фреймворків і її можна використовувати будь-де, де у вас є справа з мовою програмування Python та реляційною базою даних.

Основа Django підходу є робота з Python класами, які називаємо моделями. Кожен клас повинен унаслідуватись від базового класу “Model” та містити набір атрибутів, кожен з яких описує поле таблиці. Таким чином клас моделі описує структуру однієї таблиці в базі даних, а об’єкт даного класу відображає один запис (рядок) таблиці.

Щоб добре розібраться з матеріалом даної глави потрібно володіти мінімальними знаннями з об’єктно-орієнтованого програмування в Python. Що таке клас, об’єкт (інстанс, екземпляр класу), що таке унаслідування, ініціалізація об’єкта, атрибути і методи класу - на усі ці питання вам потрібно знати відповідь, щоб краще зрозуміти увесь наступний матеріал. Якщо вам бракує теоретичної бази, тоді зробіть паузу на вивчення основ ООП в Python.

Ось швидкий приклад Django моделі, що описує таблицю із продуктами:

¹⁵²<http://uk.wikipedia.org/wiki/SQLAlchemy>

Django модель продукту

```
1 from django.db import models
2
3 class Product(models.Model):
4
5     title = models.CharField(
6         max_length=256,
7         blank=False,
8         verbose_name="Product Title")
9
10    price = models.IntegerField(
11        blank=False,
12        verbose_name="Product Price",
13        default=0)
```

Усе необхідне для роботи з Django моделями лежить в пакеті “django.db”. Визначивши клас `Product`, який унаслідується від “`Model`”, ми отримуємо цілий набір функціоналу з доступу та управління таблицею продуктів. Таким чином після синхронізації даної моделі `Product` з базою даних, в базі даних ми отримаємо нову таблицю для продуктів. В цій таблиці, окрім поля первинного ключа (ID, Primary Key), ми матимемо стрічкове поле (`CharField`) “`title`” (назва продукту) та ціле число (`IntegerField`) “`price`” (ціна продукту).

При першому завантаженні нашого класу `Product` та синхронізації бази даних Django створить таблицю в базі даних. При цьому Django ORM підготує та запустить подібний до наступного SQL запит:

Створення таблиці в базі

```
1 CREATE TABLE demoapp_product (
2     "id" integer AUTO_INCREMENT NOT NULL PRIMARY KEY,
3     "title" VARCHAR(256) NOT NULL,
4     "price" integer DEFAULT 0 NOT NULL
5 );
```

В даному SQL запиті назва таблиці складається з назви аплікації (demoapp) та назви класу моделі продукта (product) розділеними нижнім підкресленням. Так по-замовчуванню Django ORM називає таблиці моделей.

Поле “id” є цілим числом, що збільшується автоматично на одиницю з кожним новим рядком (записом) в таблиці. Воно не може бути NULL, що означає порожнє значення. PRIMARY KEY означає, що дане поле є унікальним і по ньому можна однозначно отримувати рядок з таблиці.

Поле “title” є стрічкою з максимальною довжиною в 256 символів і також не може бути порожнім в базі.

Останнім полем буде створено поле “price”, яке також є цілим числом. Якщо при додаванні нового запису в дану таблицю не буде передане поле ціни, тоді воно автоматично встановиться у нуль. Це завдяки параметру “DEFAULT 0”.

Django ORM надає цілий ряд типів полів, які перевищують по кількості типи в базі даних. Повний список вбудований в Django ORM полів можна знайти [тут¹⁵³](#).

Кожне із полів задекларованих в класі моделі має цілий ряд параметрів для конфігурації. Частина цих параметрів безпосередньо впливає на структуру таблиці в базі (default, сам тип поля, null, max_length і т.п.), інша ж частина має відношення суто до Django функціоналу як от адміністративної частини (blank, verbose_name, і т.п.).

Поля використовуються не лише для опису структури таблиці в базі даних, але й для формування форм та полів (віджетів) всередині цих форм. Форми

¹⁵³<http://djbook.ru/rel1.7/ref/models/fields.html>

використовуються в Django адміністративній частині для управління даними в базі.

Таким чином унаслідувавши свій Python клас від “Model” класу ми автоматично отримали увесь арсенал Django ORM системи, який у наступних секціях дозволить з легкістю створювати таблиці та набивати їх даними. І все це лише маніпулюючи Python класами та об'єктами.

Модель студента

Досить теорії, переходимо до практики! Першою реалізованою моделлю буде модель студента, адже студент є основною одиницею даних у нашій аплікації.

Згідно специфікації маємо наступний список полів студента:

- ім'я;
- прізвище;
- по-батькові;
- дата народження;
- фото студента;
- номер студентського білету;
- група;
- та додаткові нотатки.

Усі поля крім по-батькові, фото та нотаток, є обов'язковими.

Поле “група” є свого роду посилання на рядок із таблиці груп. Але оскільки ми ще не реалізували моделі групи, тому залишимо дане поле на пізніше.

Відкриваємо `models.py` модуль, що знаходиться в корені аплікації `students` і додаємо клас моделі студента:

Додаємо клас студента в models.py

```
1 # -*- coding: utf-8 -*-
2
3 from django.db import models
4
5 # Create your models here.
6 class Student(models.Model):
7     """Student Model"""
8
9     first_name = models.CharField(
10         max_length=256,
11         blank=False,
12         verbose_name=u"Ім'я")
13
14     last_name = models.CharField(
15         max_length=256,
16         blank=False,
17         verbose_name=u"Прізвище")
18
19     middle_name = models.CharField(
20         max_length=256,
21         blank=True,
22         verbose_name=u"По-батькові",
23         default=' ')
24
25     birthday = models.DateField(
26         blank=False,
27         verbose_name=u"Дата народження",
28         null=True)
29
30     photo = models.ImageField(
31         blank=True,
32         verbose_name=u"Фото",
33         null=True)
```

```
35     ticket = models.CharField(  
36         max_length=256,  
37         blank=False,  
38         verbose_name=u"Білет")  
39  
40     notes = models.TextField(  
41         blank=True,  
42         verbose_name=u"Додаткові нотатки")
```

Пройдемось по найважливіших рядках:

- 1ий: utf-8 - додаємо заголовок про кодування файлу, щоб можна було використовувати кирилицю в стрічках далі у модулі;
- 3ий: import models - Django ORM живе в пакеті django.db; модуль models дає нам доступ до усього: базових класів моделей, полів, констант та конфігурації пов'язаної із базою даних;
- 6ий: class - називаємо наш клас Student і унаслідуємось обов'язково від Model класу; завдяки йому ми матимемо усю силу Django ORM використовуючи наш клас; далі побачимо, які методи та атрибути унаслідує клас Student із класу Model;
- 7ий: як завжди намагаємось тримати наш код коментованим;
- 9ий: first_name - наше перше поле; це стрічкове поле ([CharField¹⁵⁴](#)), що в базі MySQL відповідає полю VARCHAR; назва атрибута (first_name) буде використана для назви поля в таблиці бази даних;
- 10ий: [max_length¹⁵⁵](#) - це обов'язковий параметр поля CharField, який визначає максимальну довжину стрічки дозволеної у даному полі;
- 11ий: [blank¹⁵⁶](#) - на рівні форм управління об'єктами даної моделі, дане поле буде обов'язковим до заповнення;
- 12ий: [verbose_name¹⁵⁷](#) - вказує на стрічку, під якою представляти дане поле на користувачькому інтерфейсі; якщо даний параметр не вказаний, використовуватиметься назва атрибуту поля;

¹⁵⁴<http://djbook.ru/rel1.7/ref/models/fields.html#charfield>

¹⁵⁵http://djbook.ru/rel1.7/ref/models/fields.html#django.db.models.CharField.max_length

¹⁵⁶<http://djbook.ru/rel1.7/ref/models/fields.html#blank>

¹⁵⁷<http://djbook.ru/rel1.7/ref/models/fields.html#verbose-name>

- 23ий: `default158` - корисний параметр у випадку, якщо поле не є обов'язковим; якщо на формі управління об'єкта моделі поле залишено не заповненим, тоді Django ORM автоматично передасть значення із параметра `default` в якості значення поля в базі даних;
- 25ий: `birthday` - дата народження використовує поле типу `DateField159`, що вказує на дату; дане поле відповідає типу DATE в базі даних MySQL;
- 28ий: `null160` - тут ми вказуємо нашій таблиці, що поле `birthday` може мати порожнє значення;
- 30ий: `photo` - дане поле є типом `ImageField161`; воно міститиме для нас зображення студента; насправді сам файл зображення Django не зберігає в базі, а у файловій системі в папці `MEDIA_ROOT` (далі буде більше деталей про дану папку), а в базі зберігається лише назва завантаженого файла;
- 40ий: `notes` - навідміну від `CharField`, `TextField162` дозволяє містити багатострічковий текст і не потребує фіксованої довжини; на формах дане поле представлене тегом `textarea`, який є багатострічковим полем для вводу тексту.

null vs blank: досить часто путають дані два параметри полів в моделях. Насправді вони мають різну мету. `null` конфігурує базу даних вказуючи на те, чи може бути поле таблиці порожнім. В той час як `blank` вказує Django на те, чи поле на формі управління моделлю буде обов'язковим чи ні. Зазвичай вони працюють в парі і якщо `blank` є `True` (поле необов'язкове на формі), тоді `null` також є `True` (поле може бути порожнім в базі даних).

¹⁵⁸<http://djbook.ru/rel1.7/ref/models/fields.html#default>

¹⁵⁹<http://djbook.ru/rel1.7/ref/models/fields.html#django.db.models.DateField>

¹⁶⁰<http://djbook.ru/rel1.7/ref/models/fields.html#null>

¹⁶¹<http://djbook.ru/rel1.7/ref/models/fields.html#django.db.models.ImageField>

¹⁶²<http://djbook.ru/rel1.7/ref/models/fields.html#django.db.models.TextField>

Поле ImageField

Як вже було згадано вище, Django ORM не зберігає файлів в базі. Це може знизити швидкодію бази даних. Натомість бінарні файли зберігаються у файловій системі.

В налаштуваннях проекту (модуль `settings.py`) потрібно вказати абсолютний шлях до папки, де будуть зберігатись усі файли, що відносяться до даних аплікації. Ця змінна називається “`MEDIA_ROOT`”. Подібним чином треба налаштовувати URL адресу, яка обслуговуватиме дані зображення в браузері. Додаємо наступні рядки до `settings.py` модуля:

Конфігуруємо media файли

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

Таким чином в браузері під адресою “`http://localhost:8000/media/`” можна буде отримувати файли, що лежатимуть у медіа директорії. А папку із файлами у файловій системі ми покладемо на рівень вище, ніж корінь проекту, в папку “`media`”. Медіа файли, так само як і база даних, не повинні потрапляти в репозиторій коду.

Пам'ятаєте? Ми поклали зображення студентів в папку `static`. Так от, тепер ми ці зображення зберігатимемо в папці спеціально призначений для файлів, що мають відношення до даних, а не до функціоналу програми. В папці “`media`”.

Також для роботи поля `ImageField` необхідна Python бібліотека `Pillow`¹⁶³. Данна бібліотека дозволяє працювати із зображеннями та модифікувати їх з допомогою Python коду.

¹⁶³<http://pillow.readthedocs.org/en/latest/>

Файл requirements.txt

До цього моменту ми інсталювали усі додаткові Python пакети кожен окремо з допомогою команди: “`pip install [package name]`”.

Проте pip може приймати параметром файл із списком пакетів для інсталяції в Python. Оскільки маємо заінсталити Pillow, скористаємось моментом і створимо такий файлік із переліком усіх Python пакетів, що необхідні нам на даний момент для проекту.

Для цього створимо файл requirements.txt в корені нашого репозиторію:

Створюємо requirements.txt

```
1 # заходимо в корінь репозиторію
2 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb
3
4 # створюємо файл requirements.txt
5 $ touch requirements.txt
```

Відкриваємо його в редакторі і додаємо три важливі для нас на даний момент Python пакети:

Список необхідних бібліотек

```
1 Django==1.7.1
2 Pillow
3 MySQL-python
```

Як бачите в цьому файлі можна не лише перечисляти необхідні для проекту бібліотеки, але й, при потребі, фіксувати їхні версії. Так ми вказали, що нам потрібний Django саме 1.7.1 версії. Pillow потрібен для роботи з полями зображень. MySQL-python - пригодиться пізніше для переїзду із sqlite на базу даних MySQL.

Тепер заінсталімо усі перечислені пакети. Не забудьте активувати віртуальне середовище перед цим:

Інсталюємо залежності проекту

```
1 (studentsdb)$ pip install -r requirements.txt
```

Якщо вищенаведена команда поламалась при інсталяції Pillow, тоді швидше за все бракує розробницьких заголовків у бібліотеках libjpeg та freetype. У такому випадку поверніться до глави “Робоче Середовище” та перегляньте інструкцію з інсталяції Python та усіх бібліотек, від яких він залежить.

Якщо ж у вас виникла помилка під час інсталяції пакета MySQL-python, тоді однією із причин може бути брак розробницьких заголовків mysql сервера. Доставити їх на Лінуксі можна наступною командою: “\$ sudo apt-get install mysql-client libmysqlclient-dev”. Після цього зробіть ще одну спробу інсталяції “requirements.txt” бібліотек.

Якщо інсталяція пакета MySQL-python і далі ламається, але тепер з помилкою “mysql_config not found”, тоді швидше за все встановлений напередодні mysql сервер не є прописаний у шляхах системи. Це можна легко віправити додавши шлях до нього у змінну середовище. Для цього вам потрібно знати де знаходиться mysql інсталяція. Команда подібна на цю допоможе віправити проблему: “export PATH=\$PATH:/usr/local/mysql/bin”. Замість даного шляху пропишіть шлях до вашої інсталяції бази, підпапки “bin”. Данна команда допоможе на системах Лінукс та Макінтош.

Лише після успішного завершення попередньої команди ми зможемо перейти до наступної секції та активувати модель студента.

Активуємо модель

Завдяки тому, що наша перша модель лежить в модулі `models.py` всередині зареєстрованої в проекті аплікації, Django фреймворк автоматично вичитає та завантажить дані про студента.

Нам залишилось зробити лише одну річ - синхронізувати Python код з базою даних. А саме, створити нову таблицю для студентів.

Синхронізація бази даних згідно оновленого коду моделей в аплікації відбувається з допомогою команди “`makemigrations`” (генерує Python код, який оновить базу даних) і команди “`migrate`” (застосовує нові міграції до бази даних). То ж спочатку створимо нашу першу міграцію:

Створюємо міграцію

```
1 # команда приймає параметром назву аплікації, чиї моделі
2 # змінились
3 (studentsdb)$ python manage.py makemigrations students
4 Migrations for 'students':
5   0001_initial.py:
6     - Create model Student
```

Результат запуску “`makemigrations`” чітко сказав нам, що був створений модуль “`0001_initial.py`” (в папці `migrations` нашої аплікації `students`), який містить код, що застосує створення моделі `Student` на базу даних. Тобто створить таблицю студентів.

Такі міграційні модулі створюються по замовчуванню в підпапці “`migrations`” папки аплікації, де були зроблені зміни моделей. Міграції потрібні:

- розробнику для синхронізації його власних змін;
- колегам, щоб отримати в базі зміни від інших розробників;
- адміністратору, щоб застосовувати зміни моделей на продакшин сервері.

Тепер можемо запустити дану міграцію:

```
1 (studentsdb)$ python manage.py migrate
```

Якщо після запуску вище наведеної команди вискачує помилка: “Can not use ImageField because Pillow is not installed”, перегляньте процес інсталяції Pillow.

Якщо ж у вас виникла помилка: “Error loading MySQLdb module”, тоді пакет MySQL-python не встановився або встановився некоректно. Перегляньте попередню секцію і подивіться, яким результатом закінчується команда інсталяції пакетів із файліка “requirements.txt”.

В результаті успішного завершення команди “migrate” ми отримаємо нову таблицю в базі даних із назвою “students_student” ([назва аплікації]_[назва моделі]).

Код міграцій, це один із винятків, коли автогенеровані файли потрібно три-мати в репозиторії коду. Крім того, ці модулі в деяких складних випадках, потрібно підправляти вручну перед їхнім використанням.

API моделі

Перед тим, як братись до використання новоствореної моделі у в’юшках і оновлювати списки студентів реальними даними з бази, давайте трохи розберемось із тим, як ми можемо створювати нового студента і зберігати його в базі, а також як шукати існуючого студента в базі.

API (Application Programming Interface) - це програмний інтерфейс аплікації. З його допомогою ми, з коду, маємо можливість взаємодіяти із сущностями тої чи іншої системи. Більшість популярних веб-сайтів, соціальних мереж пропонують не лише графічний інтерфейс для користувача (GUI, Graphical

User Interface), але й програмний інтерфейс (API) для реалізації власних аплікацій, що взаємодіють з цими сервісами.

Ще однією класною командою в арсеналі інструментів скрипта manage.py є команда “shell”. Вона запускає для нас інтерактивний інтерпретатор Python з усіма доступними Django шляхами і бібліотеками, а також створеним зв’язком із базою даних. Усі експерименти всередині “shell” будуть збережені у нашій базі.

Перед тим як переходити до прикладів роботи із Django моделями, хочу зауважити одну річ. Якщо нам потрібно робити пошук по таблиці в базі даних, щоб отримати існуючі записи, тоді ми працюємо із методами та атрибутами КЛАСУ моделі. Якщо ж нам потрібно створити новий рядок в таблиці, тоді ми працюємо із ОБ’ЄКТАМИ класу моделі.

Запускаємо “shell” команду:

```
1 (studentsdb)$ python manage.py shell
2 Python 2.7.8 (default, Aug 30 2014, 21:10:17)
3 [GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
4 Type "help", "copyright", "credits" or "license" for more information
5 n.
6 (InteractiveConsole)
7 >>>
```

Якщо команда закінчилась стрічкою запрошення для вводу (>>>), тоді все чудово і можемо продовжувати наші перші експерименти з моделлю Student:

Вивчаємо Django ORM API

```
1 # імпортуємо клас нашої моделі
2 >>> from students.models import Student
3
4 # з django.utils імпортуємо модуль timezone, який
5 # дозволить нам створювати об'єкти Python datetime із
6 # прив'язкою до часової зони; саме так вимагають часові
7 # поля в Django; у нашому випадку це birthday;
8 >>> from django.utils import timezone
9
10 # створюємо об'єкт із класу студента, передаючи йому
11 # мінімум необхідних даних: ім'я, прізвище, день
12 # народження та номер білета
13 >>> stud = Student(first_name="Vitaliy",
14                     last_name="Podoba", birthday=timezone.now(),
15                     ticket="234")
16
17 # таким чином можемо доступатись до атрибутів
18 # (полів) студента
19 >>> stud.first_name
20 'Vitaliy'
21
22 # новостворений об'єкт попаде в таблицю даних лише після
23 # використання методу save на об'єкті студента
24 >>> stud.save()
25
26 # тепер наш об'єкт студента автоматично при збереженні в
27 # базу отримав атрибут id; порядковий номер рядка в
28 # таблиці; в моєму випадку це тип Long Integer;
29 >>> stud.id
30 1L
31
32 # тепер скористаємось атрибутом objects класу моделі, щоб
33 # отримати список усіх існуючих записів таблиці студентів
34 # з бази;
```

```
35
36 # завдяки унаслідуванню класу Model наш клас Student
37 # отримав атрибут objects, який дає нам цілий ряд методів
38 # для роботи з базою, зокрема пошук об'єктів в базі;
39
40 # метод all повертає усі рядки таблиці; зауважте, ми
41 # викликаємо атрибут objects на класі Student, а не на
42 # об'єкти класу;
43 >>> Student.objects.all()
44 [<Student: Student object>]
45
46 # метод all повернув нам список об'єктів класу Student;
47 # але це не простий список, а об'єкт типу QuerySet; він
48 # має майже увесь набір тих самих методів, що має
49 # "objects", таким чином пожемо ланцюжком викликати
50 # різноманітні фільтруючі методи один за одним:
51 >>> Student.objects.all().all().filter(id=1)
52 [<Student: Student object>]
53
54 # ми також можемо тепер отримати нашого студента з бази і
55 # оновити кілька атрибутів і потім знову пере-зберегти:
56 >>> stud = Student.objects.filter(first_name="Vitaliy")[0]
57
58 # цього разу ми використали метод filter менеджера об'єкта,
59 # який повернув нам QuerySet із одного елемента, який ми
60 # успішно зберегли в змінну stud;
61
62 # змінимо ім'я студента
63 >>> stud.first_name = "Oleg"
64
65 # і збережемо
66 >>> stud.save()
67
68 # тепер витягнемо цього студента з бази та перевіримо чи він дійсно
69 # має оновлене ім'я:
```

```
70 >>> Student.objects.get(pk=1).first_name
71 u'Oleg'
72
73 # а цього разу ми скористалимь методом get, який повертає
74 # не список, а єдиний об'єкт моделі Student; він повинен
75 # отримати аргументом поле, яке унікально ідентифікує
76 # рядок в таблиці; він видасть помилку, якщо об'єкта за
77 # даним критерієм немає в таблиці, а також, якщо знайдено
78 # більше, ніж один елемент в таблиці;
79
80 # давайте створимо ще одного студента і збережемо його
81 # в базі:
82 >>> stud2 = Student(first_name="Roman",
83                      last_name="Demo", birthday=timezone.now(),
84                      ticket="911")
85 >>> stud2.save()
86
87 # тепер давайте виведемо обидвох студентів з бази в
88 # алфавітному порядку посортувавши по імені:
89 >>> Student.objects.order_by('first_name')
90 [<Student: Student object>, <Student: Student object>]
91
92 # користі мало з такого виводу, так? тому давайте
93 # скористаємось так званими "ліст компрехеншенами" в Python
94 # https://docs.python.org/2/tutorial/datastructures.html#list-comprehensions:
95 >>> [s.first_name for s in
96 ...     Student.objects.order_by('first_name')]
97 [u'Oleg', u'Roman']
98
99
100 # а тепер впорядкуємо в зворотньому порядку додавши ще
101 # один метод до нашого ланцюжка запитів reverse:
102 >>> [s.first_name for s in
103 ...     Student.objects.order_by('first_name').reverse()]
104 [u'Roman', u'Oleg']
```

```
105  
106 # на завершення наших експериментів видалимо студента Roman  
107 >>> roman = Student.objects.get(first_name="Roman")  
108 >>> roman.delete()  
109 >>> len(Student.objects.all())  
110 1  
111  
112 # ми отримали студента з допомогою методу get та поля  
113 # first_name; викликали на об'єкті студента метод delete,  
114 # який видалив рядок із таблиці, що відповідав студенту  
115 # Roman;  
116  
117 # щоб переконатись, що студент був успішно видалений, ми  
118 # порахували усіх студентів в таблиці функцією len на  
119 # QuerySet об'єкті, що нам повернув методі all;  
120  
121 # для того, щоб вийти з інтерактивної сесії, скористайтесь  
122 # комбінацією клавіш Ctrl-D.
```

Таким чином, варто запам'ятати кілька важливих речей при роботі з моделями:

- для додавання нового рядка в таблицю бази потрібно створити новий об'єкт із класу моделі, передати їй дані (при створенні чи пізніше встановити через атрибути) та викликати метод save;
- для видалення існуючого рядка з таблиці бази потрібно викликати метод delete на об'єкті моделі, що представляє даний рядок;
- для отримання даних із бази необхідно використовувати менеджер об'єктів¹⁶⁴, який є доступний по-замовчуванню як атрибут "objects" на рівні класу моделі; є можливість також реалізовувати кастомні менеджери, але нам це не пригодиться;

¹⁶⁴<http://djbook.ru/rel1.7/topics/db/managers.html#django.db.models.Manager>

- основні методи менеджера об'єктів, які найчастіше використовуються, це: `all165`, `filter166`, `exclude167`, `get168`, `get_or_create169`, `order_by170`, `reverse171`, `exists172`;
- кожен із вище перечислених методів (окрім `get`) повертає множину елементів з таблиці - об'єкт типу `QuerySet173`;
- методи менеджера об'єктів, які повертають об'єкт типу `QuerySet` (множину з результатами з бази даних) також можна викликати ланцюжком один за одним для застосування одразу кількох фільтрів та умов в запиті до бази (напр. `all().filter(name="MtM").order_by("title")`);
- параметри всередині методів менеджера об'єктів можуть мати додаткові фільтри¹⁷⁴, щоб дозволити створювати гнучкіші та динамічніші запити в базу: `exact175`, `contains176`, `in177`, `gt178`, `lt179`, `startswith180`, `endswith181`, `year182`, `regex183` і ще багато інших.

Попрошу ваш ознайомитись із вище перечисленими атрибутами і методами моделей в Django для розуміння усіх можливостей потужної ORM системи. Ту частину методів та атрибутів, яку ми з вами використовуватимемо в проекті, далі будемо розбирати детальніше.

¹⁶⁵<http://djbook.ru/rel1.7/ref/models/querysets.html#all>

¹⁶⁶<http://djbook.ru/rel1.7/ref/models/querysets.html#filter>

¹⁶⁷<http://djbook.ru/rel1.7/ref/models/querysets.html#exclude>

¹⁶⁸<http://djbook.ru/rel1.7/ref/models/querysets.html#get>

¹⁶⁹<http://djbook.ru/rel1.7/ref/models/querysets.html#get-or-create>

¹⁷⁰<http://djbook.ru/rel1.7/ref/models/querysets.html#order-by>

¹⁷¹<http://djbook.ru/rel1.7/ref/models/querysets.html#reverse>

¹⁷²<http://djbook.ru/rel1.7/ref/models/querysets.html#exists>

¹⁷³<http://djbook.ru/rel1.7/ref/models/querysets.html>

¹⁷⁴<http://djbook.ru/rel1.7/topics/db/queries.html#field-lookups-intro>

¹⁷⁵<http://djbook.ru/rel1.7/ref/models/querysets.html#exact>

¹⁷⁶<http://djbook.ru/rel1.7/ref/models/querysets.html#contains>

¹⁷⁷<http://djbook.ru/rel1.7/ref/models/querysets.html#in>

¹⁷⁸<http://djbook.ru/rel1.7/ref/models/querysets.html#gt>

¹⁷⁹<http://djbook.ru/rel1.7/ref/models/querysets.html#lt>

¹⁸⁰<http://djbook.ru/rel1.7/ref/models/querysets.html#startswith>

¹⁸¹<http://djbook.ru/rel1.7/ref/models/querysets.html#endswith>

¹⁸²<http://djbook.ru/rel1.7/ref/models/querysets.html#year>

¹⁸³<http://djbook.ru/rel1.7/ref/models/querysets.html#regex>

Адмін інтерфейс

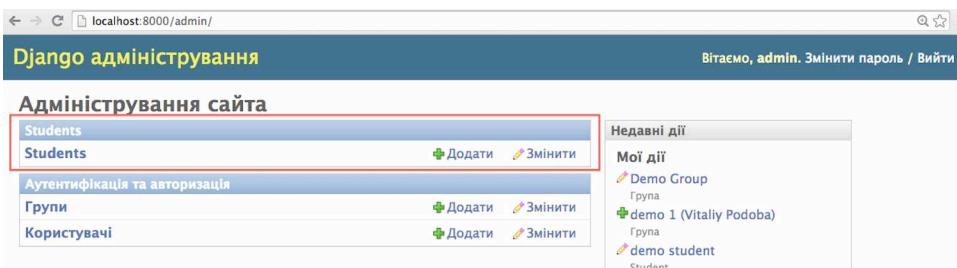
Нам залишилось зробити ще одну річ - зробити модель студента видимою для адміністративного інтерфейсу Django. По-замовчуванню новостворені моделі не видимі в адмінці.

Для того, щоб модель студента з'явилася в адмінці, необхідно оновити модуль admin.py в корені аплікації наступним рядком:

Реєструємо модуль Student для Django адмінки, admin.py

```
1 from django.contrib import admin
2 from .models import Student
3
4 # Register your models here.
5 admin.site.register(Student)
```

Функція register повідомляє Django про нашу модель і просить його додати її до адмін частини. Тепер, коли зайдете на адміністративний інтерфейс нашої аплікації (“<http://localhost:8000/admin/>”), побачите секцію студентів серед списку доступних моделей:



The screenshot shows the Django Admin dashboard at localhost:8000/admin/. The top navigation bar includes links for back, forward, search, and user information ('Вітаємо, admin. Змінити пароль / Вийти'). The main title is 'Django адміністрування'. Below it, a sidebar titled 'Адміністрування сайту' lists the 'Students' app with its 'Students' model registered. A red box highlights this entry. To the right, there's a sidebar titled 'Недавні дії' (Recent Actions) showing a list of recent changes made by the user 'admin': 'Demo Group' (Група), 'demo 1 (Vitaliy Podoba)' (Група), and 'demo student' (Студент). The bottom of the screenshot has a caption: 'Students серед моделей в адміністративній частині Django'.

Students серед моделей в адміністративній частині Django

Зауважте, що у нас з'явилася ціла нова секція Students, яка містить список моделей, що приходять із аплікації “students”. Поки там лише одна модель, але згодом матимемо і групи, і журнал відвідування.

Ви можете зайти всередину секції управління студентами клікнувши по лінку “Students”. Там отримаєте список існуючих в базі студентів, а також форми для додавання, редагування та видалення студентів.

Список Студентів в Django Admin

Як бачите маємо дві проблеми:

- на попередньому зображенні секція студентів містить лінки, що не є перекладеними на українську мову;
- із списка студентів важко зrozуміти хто є хто.

Щоб отримати слово “Студент” в адмін інтерфейсі Django всюди, де є посилання на модель студента, додамо вкладений клас `Meta184` всередині класу моделі студента:

¹⁸⁴<http://djbook.ru/rel1.7/ref/models/options.html>

Призначаємо user-friendly ім'я моделі студента в адмінці, models.py

```
1 # Create your models here.
2 class Student(models.Model):
3     """Student Model"""
4
5     class Meta(object):
6         verbose_name = u"Студент"
7         verbose_name_plural = u"Студенти"
8
9     first_name = models.CharField(
10         max_length=256,
11         blank=False,
12         verbose_name=u"Ім'я")
```

Вище наведений лише фрагмент модуля models.py, де ми додали клас Meta всередину класу Student. Так, навіть таке можна робити в Python.

Meta клас унаслідує вбудований Python клас object. Так звані класи нового типу були введені починаючи з Python версії 2.2. Вони принесли такі нові речі як декоратори, функцію super та багато інших. Якщо ваш клас не потребує інших класів для унаслідування, тоді рекомендовано унаслідуватись як мінімум від object.

В класі Meta ми визначили два атрибути: verbose_name та verbose_name_plural. Перший визначає стрічку для використання в Django адмінці, щоб позначати модель студента. Другий - модель студента у формі множини.

Після даної зміни перевантажте корінь вашої адмінки і отримаєте наступні покращення:

Клас Meta в дії: verbose_name атрибут

Таким чином клас із назвою Meta має особливий вплив на модель, в якій він знаходитьться. З його допомогою ми можемо налаштовувати поведінку моделі, яка може впливати як на структуру таблиці моделі (напр. назва таблиці моделі), так і на Django функціонал (напр. вигляд моделі в адміністративній частині).

Тепер давайте пофіксимо другу проблему і зробимо так, щоб список існуючих студентів в базі відображався в Django адмініці у вигляді: "Ім'я Прізвище".

Для цього передбачений метод “`__unicode__`” на рівні класу моделі, який повинен повернати стрічку для представлення моделі як в інтерактивній стрічці “shell”, так і в Django адміністративній частині. У даному методі, як і у будь-яких інших динамічних методах класу, ми маємо доступ до об'єкта і його даних. То ж скористаймося з цього:

Додаємо метод для гарного представлення студента, модуль `models.py`

```
1 def __unicode__(self):
2     return u"%s %s" % (self.first_name, self.last_name)
```

Додайте даний метод наприкінці тіла класу. У ньому ми віддаємо юнікову стрічку, у вигляді “[first_name] [last_name]”. Доступ до полів студента ми маємо через ключове слово `self`, що передається першим аргументом кожного динамічного метода класу в Python. Саму стрічку формуємо з допомогою так званої інтерполяції стрічок¹⁸⁵.

Спробуйте тепер перезавантажити [спісок студентів](#)¹⁸⁶:

¹⁸⁵http://www.internet-technologies.ru/articles/article_220.html

¹⁸⁶<http://localhost:8000/admin/students/student/>

Виберіть Студент щоб змінити

Дія: ----- 0 з 10 обрано

<input type="checkbox"/>	Студент
<input type="checkbox"/>	Vitaliy Podoba
<input type="checkbox"/>	Vitaliy Podoba
<input type="checkbox"/>	Danylo Test
<input type="checkbox"/>	student 3 studnet 3
<input type="checkbox"/>	demo student
<input type="checkbox"/>	student 1 last name 1
<input type="checkbox"/>	Ярослав Іздрик
<input type="checkbox"/>	Назар Кеньо
<input type="checkbox"/>	Новий Студент
<input type="checkbox"/>	Oleg Podoba

10 Студенти

Оновлений список студентів в адмінці

Спробуйте додати ще 2-3 студента і поредагувати існуючі. Та й загалом поекспериментувати із Django адмін інтерфейсом. Кілька додаткових студентів нам пригодяться далі для оновлення коду в'юшок.

На даний момент маємо усе необхідне, щоб оновити головну сторінку нашої аплікації списком студентів із бази даних!

Тепер можемо зробити коміт та пуш в код репозиторій. Модель студента практично готова і присутня в адмінці. Це можна оформити окремим комітом. І не забувайте наприкінці кожної секції глави робити коміт в репозиторій. Також пам'ятайте про хороші пояснювальні коментарі до кожного коміту.

Оновлюємо список студентів

В попередній главі книги ми з вами тимчасово запрограмували в'юшку списку студентів (`students_list`) повернати “прибитий” список словників, кожен

з яких представляє собою дані по уявному студенту. Завдяки цьому хаку ми з вами повчилися мови шаблонів Django використовуючи теги умов та циклів.

Тепер настав час оновити код цієї функції в'юшки та заставити її витягувати для нас список студентів із таблиці в базі даних. Ви не повірите, але кількість рядочків в даній функції значно зменшиться:

Повертаємо усіх студентів з бази, `students/views/students.py`

```

1 from ..models import Student
2
3 def students_list(request):
4     students = Student.objects.all()
5     return render(request, 'students/students_list.html',
6                   {'students': students})

```

Ви вже знайомі із менеджером об'єктів “objects” та його методом “all”. Замість того, щоб самостійно формувати список словників, ми просто скористалися ORM API викликами, щоб отримати усіх студентів з бази. Звичайно нам також треба було імпортувати модель студента.

Тепер перезавантажте головну сторінку нашої аплікації і ви знову не повірите - список працює і повертає реальних студентів з бази. Без будь-яких додаткових змін в самому шаблоні:

#	Фото	Прізвище	Ім'я	№ Білету	Дії
1		Podoba	Oleg	1234	Дія ▾
2		Студент	Новий	234	Дія ▾
3		Кеньо	Назар	2123	Дія ▾

Список студентів після оновлення в'юшки для роботи з базою даних

Спробуйте здогадатись, чому список загалом працює?...

Початково ми сформували прибитий список словників, де назви полів були в більшості рівні полям, які ми прописали в класі моделя студента. Тому більшість коду в шаблоні працює з новими типами даних.

Але бачите, що зображення таки поламались. З ними треба працювати по-іншому, адже тепер наші зображення є частиною полів моделі, а не просто статичними ресурсами в папці “static”.

Оновлюємо зображення студента

Ми уже знаємо, що файли та зображення в полях моделей насправді зберігаються в папці під адресою MEDIA_ROOT, а обслуговуються вони в браузері під адресою MEDIA_URL. Так от, щоб вручну не формувати посилання на зображення, Django ORM підготував властивість “url”, яку можна викликати на полі зображення і воно поверне нам шлях на зображення враховуючи MEDIA_URL налаштування.

Таким чином можемо замінити наш тег “img” в шаблоні студентів наступним тегом:

Тег зображення посилається на зображення із MEDIA_ROOT

```
1 
```

Із студента ми витягнули поле “photo”, а у поля foto викликали атрибут “url”. Він нам і поверне адресу зображення студента.

Але цього не достатньо для того, щоб наші зображення запрацювали. Навідміну від веб адреси для статичних ресурсів, веб адреса для медіа ресурсів не включена у дефолтні URL патерни в Django. Для медіа ресурсів, щоб можна їх було отримувати в браузері, потрібно додати URL шаблон, який буде постачати медіа файли:

Додаємо медіа шаблон до urls.py модуля

```
1 from .settings import MEDIA_ROOT, DEBUG
2
3 if DEBUG:
4     # serve files from media folder
5     urlpatterns += patterns('',
6         url(r'^media/(?P<path>.*)$', 'django.views.static.serve', {
7             'document_root': MEDIA_ROOT}))
```

Вище наведений кусок коду додаєте наприкінці модуля urls.py. Аналізуючи порядково даний код, можемо відмітити основні моменти:

- 1ий: імпортуємо кілька констант з нашого модуля налаштувань проекту; вони нам пригодяться далі;
- 3ий: ми обслуговуємо медіа файли через Django фреймворк лише під час розробки; на продакшин вебсайтах статичні та медіа ресурси обслуговуються напряму через кінцевий сервер (напр. Apache або Nginx); тому “media” патерн додаємо лише, коли режим DEBUG є включеним;
- 5ий: додаємо до уже визначених патернів ще один набір патернів; шаблони можна просто додавати один до одного;
- 6ий: шляхи типу “/media/filename” будуть обслуговуватись обробником “serve”, що приходить із модуля статичних в’юшок; це той самий обробник, що працює із статичними ресурсами; назва файла передаватиметься йому під аргументом “path”;
- 7ий: окрім того, що в’юшка може отримувати в якості аргументів групи з регулярних виразів URL шаблона, її також можна передавати наперед заготовані аргументи через словник; саме таким чином ми передаємо аргумент ‘document_root’, що рівний шляху до кореневої папки з медіа ресурсами; якщо пам’ятаєте, ми встановили даний параметр в settings.py в попередніх секціях.

Спробуйте тепер перезавантажити сторінку із списком студентів і глянути на результат.

Якщо зображення з'явились і все працює добре, тоді вам просто пощастило. У вашій базі усі студенти мають встановлене зображення. Поверніться в адмін частину і спеціально видаліть зображення в одного із студентів. Після цього знову оновіть сторінку із списком студентів. Отримаєте помилку: “The ‘photo’ attribute has no file associated with it.” Данна помилка вказує на те, що один із студентів немає зображення встановленого, відповідно виклик атрибута “url” ламається.

Поле “photo” студента не є обов’язковим для заповнення, тому логічним буде припустити, що не всі студенти матимуть зображення і наша аплікація ламатиметься у таких випадках.

Щоб пофіксити дану проблему, як один із варіантів, можемо вставляти фото студента, лише при умові, що поле непорожнє. А коли поле порожнє, виводити дефолтне зображення аноніма. Наприклад:

Показуємо зображення студента лише, коли воно встановлене

```
1      {% if student.photo %} 
2          
5      {% else %} 
6          
9      {% endif %}
```

В першому рядку ми перевіряємо чи студент має поле foto не порожнє. Якщо поле “photo” встановлено, тоді використовуємо його адресу в якості атрибута “src” для тегу “img”.

В рядку 5 ми розпочинаємо гілку коду (else) у випадку, якщо foto студента не встановлено. У цьому випадку ми використовуємо зображення із статичної папки “static/img” під назвою “default_user.png”.

Я просто в Гуглі пошукав перше краще зображення анонімного користувача за ключовим словом “default user image” та включив закладку Зображення і обрав невеликого розміру підходящий малюнок. Можете також знайти для себе подібне зображення, або скористатись тим, яке знайдете в коді проекту, що йде разом із книгою. Скопіюйте його собі в підпапку “static/img” в аплікації.

Ось результат наших старань:

Сервіс Обліку Студентів

Група: Усі Студенти

Студенти Відвідування Групи

База Студентів

Додати Студента					
#	Фото	Прізвище	Ім'я	№ Білету	Дії
1		Podoba	Oleg	1234	<button>Дія ▾</button>
2		Студент	Новий	234'	<button>Дія ▾</button>
3		Кеньо	Назар	2123	<button>Дія ▾</button>
4		Iazdrick	Ярослав	999	<button>Дія ▾</button>

Зображення студентів працюють!

Зображення студентів працюють. А для тих, в кого його поки ще немає, ми показуємо зображення анонімного користувача.

Навігація та сортування списку студентів

Перед тим як переходити до розробки моделі Групи, повправляємось трохи в шаблонах та API Django моделей. Покращимо список студентів двома додатковими речима:

- сортування рядків таблиці по колонках: ім'я, прізвище та номер білета; в обидві сторони: по зростанню та спаданню;

- динамізуємо посторінкову навігацію по студентах, де показуватимемо 3 студента на одній сторінці.

Зазвичай кількість елементів на сторінці при посторінковій навігації визначається більшим числом, яке буде золотою серединою між довгим скролом на одній сторінці та великою кількістю навігаційних кліків. 10-50 студентів на сторінці у нашому випадку буде резонним числом. Але для тестування ми оберемо лише 3 студента на сторінку, щоб не прийшлося довго набивати тестові дані в базі.

Сортуємо по колонках

Спочатку розберемось із сортуванням. А почнемо із оновлення серверної сторони.

Ми хочемо сортувати по трьох різних полях студента, а також в зворотньому порядку. Тому із клієнта на сервер повинна приходити інформація про поле та няпрамок. Її можемо передати у вигляді додаткових параметрів запиту в URL адресі і нехай:

- `order_by` буде передавати нам назву поля, по якому потрібно сортувати;
- а `reverse` із значенням 1 буде повідомляти нашу в'юшку, що потрібно сортувати в зворотньому порядку.

Оновлена в'юшка списку студентів виглядатиме наступним чином:

students_list із сортуванням студентів

```
1 def students_list(request):
2     students = Student.objects.all()
3
4     # try to order students list
5     order_by = request.GET.get('order_by', '')
6     if order_by in ('last_name', 'first_name', 'ticket'):
7         students = students.order_by(order_by)
8         if request.GET.get('reverse', '') == '1':
9             students = students.reverse()
10
11    return render(request, 'students/students_list.html',
12                  {'students': students})
```

Блок із шістьох рядочків коду зробив для нас усю справу:

- 5ий рядок: перевіряємо чи ми отримали ключ ‘order_by’ в GET словнику запиту; на сторінці матимемо залінковані заголовки таблиці, що включатимуть параметри сортування; клік по лінку на веб-сторінці це завжди запит типу GET, тому параметри запиту (частина, що йде після знака питання) потрапляють саме в словник GET;
- 6ий: перевіряємо чи значення параметру ‘order_by’ є серед дозволених для сортування полів; якщо ні, тоді просто ігноруємо блок сортування студентів;
- 7ий: власне саме сортування студентів; order_by метод приймає назву поля, а повертає знову ж таки об’єкт типу QuerySet, над яким ми знову можемо робити подальші запити;
- 8ий: якщо в запиті ми також знаходимо ключ ‘reverse’ і він рівний одиниці;
- 9ий: тоді ми застосовуємо ще одну модифікацію до QuerySet студентів - метод reverse; він поєднує студентів у зворотньому порядку.

Ось так доволі просто ми додали сортувалку студентів до нашої в’юшки.

Щоб перевірити, що наш код працює правильно ще до того як ми оновили шаблон, можна просто в браузері ввести посилання: “http://localhost:8000/?order_by=ticket&reverse=1” і переконатись, що рядки таблиці будуть посортовані по назві білета і в спадаючому порядку. Можете також поекспериментувати із іншими полями, що годяться для сортування.

Тепер час оновити наш шаблон із таблицею студентів, щоб можна було сортувати студентів не лише при вводі складних URL адрес в рядок адреси браузера, але й при кліку по відповідних заголовках таблиці.

Ми хочемо, щоб клік по заголовку таблиці “Ім’я” сортував рядки студентів згідно їхнього імені в алфавітному порядку. Після кліку справа від тексту “Ім’я” повинна з’являтись стрілочка вгору. Вона буде індикатором для користувача, що в даний момент студенти посортовані по іменах. При повторному кліку по заголовку “Ім’я” ми сортуватимемо студентів по імені, але тепер у зворотньму порядку: від Я до А. А замість стрілки вгору будемо мати стрілку вниз в якості індикатора реверсного сортування.

Почнемо із найпростішого: огорнемо наші заголовки таблиць лінками, що мають в собі параметр `order_by`, а все решта будемо додавати покроково. Завдання не дуже просте, тому якщо спробувати зробити все й одразу, можна запутатись:

Додаємо `order_by` параметр до лінків в заголовках таблиці

```
1  <thead>
2      <tr>
3          <th>#</th>
4          <th>Фото</th>
5          <th><a href="{% url "home" %}?order_by=last_name">Прізвище</a>\n</th>
6      </th>
7          <th><a href="{% url "home" %}?order_by=first_name">Ім'я</a></th>
8      <th>
9          <th><a href="{% url "home" %}?order_by=ticket">№ Білету</a></th>
10     <th>Дії</th>
11     </tr>
12 </thead>
```

Я не дублюю усього шаблона з метою економії місця та, щоб книга не розрослась до тисячі сторінок. Думаю в даний момент ви уже можете з легкістю орієнтуватись в шаблоні та співставляти наведені куски коду із місцем в шаблоні. Аналогічно і з Python модулями.

У вище наведеному куску коду із заголовком таблиці ми динамізували адреси лінків заголовків: “Ім’я”, “Прізвище” та “№ Білету”. Скористалися для цього уже відомим нам тегом “url” та додали до нього параметр `order_by` із назвою відповідного поля студента.

Спробуйте перевантажити сторінку і поклікати по оновлених заголовках таблиці. Ніби працює, але є одне але. Я б сказав, що їх є два:

- важко зрозуміти по якому стовпчику відбувається сортування; тобто бракує стрілки-індикатора;
- поки неможливо посортувати у зворотньому порядку.

Давайте додамо стрілку вгору до заголовків таблиць, по яких відбувається сортування:

Додаємо стрілку вгору

```
1  <thead>
2      <tr>
3          <th>#</th>
4          <th>Фото</th>
5          <th>
6              <a href="{% url "home" %}?order_by=last_name">
7                  Прізвище
8                  {% if request.GET.order_by == 'last_name' %}&uarr;{% endif\
9                  %}
10             </a>
11         </th>
12         <th>
```

```

13         <a href="{% url "home" %}?order_by=first_name">
14             Ім'я
15             {% if request.GET.order_by == 'first_name' %}&uarr;{% endif \
16 f %}
17         </a>
18     </th>
19     <th>
20         <a href="{% url "home" %}?order_by=ticket">
21             № Білету
22             {% if request.GET.order_by == 'ticket' %}&uarr;{% endif %}
23         </a>
24     </th>
25     <th>Дії</th>
26     </tr>
27 </thead>

```

В рядку номер 8 ми додали HTML елемент “↑”, який вставляє символ стрілки вгору одразу після тексту “Прізвище”. Але дана стрілка буде вставлятись на сторінку лише, якщо в запиті присутній параметр ‘order_by’ і він рівний ‘last_name’. Тобто, лише якщо в даний момент таблиця була посортована по прізвищу. Таку ж зміну ми проробили із заголовками “Ім’я” та “№ Білету”.

Якщо тепер спробуєте оновити сторінку і клацнути заголовок “Ім’я”, то повинні отримати подібну картинку:

База Студентів					
Додати Студента					
#	Фото	Прізвище	Ім'я ↑	№ Білету	Дії
1		Test	Danylo		<button>Дія ▾</button>
2		student	demo	456	<button>Дія ▾</button>
3		Podoba	Oleg	1234	<button>Дія ▾</button>

Сортування по імені

Залишилась єдина річ - при повторному клацанні на посортований заголовок сортувати в зворотньому порядку. Для цього треба додати параметр ‘reverse’ рівний одиниці. Але додавати його треба лише, якщо:

- в даний момент уже є обраний для сортування поточний заголовок;
- в даний момент уже не відбувалось сортування в зворотньому порядку.

Усю цю умову додамо в параметр ‘reverse’ та стрілку вниз, яка замінить стрілку вгору при зворотньому сортуванню:

Додаємо зворотнє сортування і стрілку вниз

```
1 <thead>
2   <tr>
3     <th>#</th>
4     <th>Фото</th>
5     <th>
6       <a href="{% url "home" %}?order_by=last_name{% if request.GET\>
7 T.order_by == 'last_name' and request.GET.reverse != '1' %}&amp;reve\>
8 rse=1{% endif %}">
9       Прізвище
10      {% if request.GET.order_by == 'last_name' and request.GET.\>
11 reverse != '1' %}&uarr;
12      {% elif request.GET.order_by == 'last_name' and request.GET.\>
13 T.reverse == '1' %}&darr;
14      {% endif %}
15     </a>
16   </th>
17   <th>
18     <a href="{% url "home" %}?order_by=first_name{% if request.G\>
19 ET.order_by == 'first_name' and request.GET.reverse != '1' %}&amp;re\>
20 verse=1{% endif %}">
21     Ім'я
22     {% if request.GET.order_by == 'first_name' and request.GET.\>
23 .reverse != '1' %}&uarr;
24     {% elif request.GET.order_by == 'first_name' and request.G\>
25 ET.reverse == '1' %}&darr;
26     {% endif %}
27   </th>
28   <th>
```

```

29         <a href="{% url "home" %}?order_by=ticket{% if request.GET.o\
30 rder_by == 'ticket' and request.GET.reverse != '1' %}&reverse=1{\\
31 % endif %}">
32             № Білету
33             {% if request.GET.order_by == 'ticket' and request.GET.rev\
34 erse != '1' %}&uarr;
35             {% elif request.GET.order_by == 'ticket' and request.GET.r\\
36 everse == '1' %}&darr;
37             {% endif %}
38         </th>
39         <th>Дії</th>
40     </tr>
41 </thead>

```

Як бачите умову і логіка загалом значно ускладнилась. З допомогою кастомних тегів, фільтрів, процесорів контексту та в'юшок можна і зазвичай варто цього уникати. Але в даний момент ми цим займатись не будемо.

Давайте на прикладі заголовка “Прізвище” розберемо що ж змінилось:

- до самого лінка ми додали параметр `reverse="1"`, але робимо це лише тоді, коли таблиця уже посортована по прізвищу (тобто маємо в запиті ключ “`order_by`” рівний значенню “`last_name`”) і коли “`reverse`” не є уже включеним;
- умову вставки стрілки вгору ми ускладнили ще одним виразом в умові: перевіркою чи часом не застосоване уже реверсне сортування; якщо так - стрілка вгору не показується;
- якщо сортування по Прізвищу уже застосоване, а також реверс включений, тоді замість стрілки вгору показуємо стрілку вниз.

І така сама логіка додана до усіх сортувальних заголовків таблиці. Оновіть відповідно код у вашому шаблоні та перевантажте сторінку. При повторному кліку на заголовки маєте бачити зміну стрілочок, ну і звісно сортування рядків відповідно до обраного методу сортування.

На закінчення даної під-секції можна спробувати трохи упростити довжелезні вирази умов в тегах “if”. В кожному із тегів ми повторно дістаємо параметри “order_by” та “reverse” із запиту. Натомість можемо скористатись тегом присвоєння змінних (це так само як присвоювати змінну в мові Python) і уникнути дублювання коду та спростити умови:

Спрощуємо умови використовуючи тег with

```
1   {% with order_by=request.GET.order_by reverse=request.GET.reverse \
2   %}
```

```
3   <thead>
4     <tr>
5       <th>#</th>
6       <th>Фото</th>
7       <th>
8         <a href="{% url "home" %}?order_by=last_name{% if order_by =\
9 = 'last_name' and reverse != '1' %}&reverse=1{% endif %}">
10        Прізвище
11        {% if order_by == 'last_name' and reverse != '1' %}&uarr;
12        {% elif order_by == 'last_name' and reverse == '1' %}&darr;
13        {% endif %}
14      </a>
15    </th>
16    <th>
17      <a href="{% url "home" %}?order_by=first_name{% if order_by \
18 == 'first_name' and reverse != '1' %}&reverse=1{% endif %}">
19        ІМ'я
20        {% if order_by == 'first_name' and reverse != '1' %}&uarr;
21        {% elif order_by == 'first_name' and reverse == '1' %}&dar\
22 r;
23        {% endif %}
24    </th>
25    <th>
26      <a href="{% url "home" %}?order_by=ticket{% if order_by == '\
27 ticket' and reverse != '1' %}&reverse=1{% endif %}">
28        № Білету
```

```

29      {% if order_by == 'ticket' and reverse != '1' %}&uarr;
30      {% elif order_by == 'ticket' and reverse == '1' %}&darr;;
31      {% endif %}
32    </th>
33    <th>Дії</th>
34  </tr>
35 </thead>
36 {% endwith %}

```

Тегом `with187` ми огорнули усю частину коду, де потребуємо доступу до двох нових змінних:

- `order_by`, якій призначили ключ ‘`order_by`’ із запиту;
- і `reverse`.

Маючи доступні ці дві змінні, ми, у вище наведеному коді, замінили усі звернення до “`request.GET`” використанням “`order_by`” та “`reverse`”. Як бачите код значно спростиився. Принаймні його кількість зменшилась.

На домашнє завдання: додати сортування по першій колонці таблиці. А також при першому завантаженні сторінки автоматично сортувати по Прізвищу студента. Відповідно стрілочка вгору повинна з’явитись одразу після заголовка “Прізвище”.

Із сортуванням розправились! На порядку денному - посторінкова навігації:

Посторінкова навігація

У нас є вже статична заготовка посторінкової навігації внизу таблиці із студентами, яку ми швидко додали використовуючи Twitter Bootstrap віджет:

¹⁸⁷ <http://djbook.ru/rel1.7/ref/templates/builtins.html#with>



Посторінкова навігація студентів

Потрібно її динамізувати. Це завдання також складається із двох частин:

- оновити серверну сторону, щоб повертала студентів частинами, відповідно до номера поточної сторінки;
- оновити шаблон студентів, де динамізувати лінки посторінкової навігації.

Як взагалі повинна працювати дана посторінкова навігація?

На кожній сторінці повинна показуватись обмежена фіксована кількість студентів (до прикладу візьмемо 3). При навігації по сторінках на сервер передається параметр, що вказує потрібний номер сторінки. Сервер, маючи номер даної сторінки, віддає не увесь список студентів, а лише той кусок списку, що відповідає даній сторінці. Крім того, разом із списком, він передає іншу інформацію шаблону: кількість сторінок, наявністю попередньої та наступної сторінок. Все це є необхідним, щоб динамізувати навігаційний віджет на сторінці.

Отже, нехай наш параметр сторінки буде називатись “page”. Він передаватиме номер сторінки, яку потрібно відобразити. Писати усю логіку із поділом списку на групи, що відповідають сторінкам, ми не будемо. Натомість скористаємося вбудованим в Django пакетом `django.core.paginator`¹⁸⁸. Він дасть нам усю необхідну логіку і нам залишиться лише використати правильні методи:

¹⁸⁸<http://djbook.ru/rel1.7/topics/pagination.html>

Додаємо логіку з посторінкової навігації у в'юшку students_list

```
1 from django.shortcuts import render
2 from django.http import HttpResponseRedirect
3 from django.core.paginator import Paginator, EmptyPage, \
4     PageNotAnInteger
5
6 from ..models import Student
7
8
9 def students_list(request):
10     students = Student.objects.all()
11
12     # try to order students list
13     order_by = request.GET.get('order_by', '')
14     if order_by in ('last_name', 'first_name', 'ticket'):
15         students = students.order_by(order_by)
16         if request.GET.get('reverse', '') == '1':
17             students = students.reverse()
18
19     # paginate students
20     paginator = Paginator(students, 3)
21     page = request.GET.get('page')
22     try:
23         students = paginator.page(page)
24     except PageNotAnInteger:
25         # If page is not an integer, deliver first page.
26         students = paginator.page(1)
27     except EmptyPage:
28         # If page is out of range (e.g. 9999), deliver
29         # last page of results.
30         students = paginator.page(paginator.num_pages)
31
32     return render(request, 'students/students_list.html',
33                  {'students': students})
```

Окрім того, що ми додали імпорт нових класів з пакету “paginator”, в нашій функції в’юшки також з’явився новий блок коду “paginate students”. Давайте розберемо його в деталях:

- Зій: імпортуємо необхідні класи для роботи з paginator пакетом; `Paginator189`
 - клас, який організує нам всю логіку з формування студентів частинами, відповідно до кількості елементів на одній сторінці; `EmptyPage`
 - помилка, яку віддає метод “page” класу Paginator у випадку, якщо йому передали неіснуючий номер сторінки; `PageNotAnInteger` - також клас помилки, який віддає метод “page”, але вже у випадку, якщо йому передали агументом дані, які не можна перетворити в ціле число; обидвома помилками ми скористаємося при формуванні сторінки далі у функції в’юшки;
- 20ий: створюємо об’єкт Paginator класу; він потребує два обов’язкових параметра: список елементів, кількість елементів на одній сторінці;
- 21ий: отримуємо параметр ‘page’ із запиту; так, його ми знову ж таки передаватимемо параметром в GET запиті подібно до того, як ми це робили із параметрами сортування колонок таблиці;
- 22ий: усю генерацію сторінки ми огортаємо в оператор перехоплення помилок, адже ця генерація сторінки видає помилки у випадку, якщо передано неправильні аргументи, або номери неіснуючих сторінок;
- 23ій: основна логіка із формування фрагмента студентів для сторінки закладена в методі `page190` класу Paginator; даний метод отримує номер сторінки, яку потрібно згенерувати, а повертає об’єкт типу `Page191`, даний об’єкт працює в якості списку елементів, по якому можна пробігатись; окрім того він має цілий набір додаткових атрибутів, що дадуть нам усю необхідну інформацію в шаблоні для динамізації віджета посторінкової навігації;
- 24ий: якщо параметр ‘page’ не можна перевести у ціле число, тоді метод `page` викине помилку `PageNotAnInteger`; в такому випадку ми просто згенеруємо першу сторінку навігації як продемонстровано в рядку 25;

¹⁸⁹<http://djbook.ru/rel1.7/topics/pagination.html#django.core.paginator.Paginator>

¹⁹⁰<http://djbook.ru/rel1.7/topics/pagination.html#django.core.paginator.Paginator.page>

¹⁹¹<http://djbook.ru/rel1.7/topics/pagination.html#page-objects>

- 27ий: якщо ж параметр ‘page’ можна перевести у ціле число, але дане число не відповіде жодній із сторінок, які можна сформувати із списку студентів (від’ємне число, або число більше, ніж загальна кількість доступних сторінок), метод page видасть помилку типу EmptyPage; в такому випадку ми генеруємо список студентів для останньої сторінки навігації;
- 30ий: генеруємо останню сторінку навігації; ми скористалися атрибутом об’єкта Paginator “num_pages”; він повертає кількість сформованих сторінок навігації.

Таким чином об’єкт Paginator віддав нам об’єкт сторінки Page, який дає нам цілий ряд корисних методів та атрибутів, якими ми скористаємося далі в шаблоні. Також об’єкт типу Page має доступ до об’єкта Paginator через власний атрибут “paginator”. Тому із шаблона маємо доступ і до методів Paginator і до методів об’єкта Page.

Для швидкого тесту і підтвердження того, що наша логіка серверна працює, перевантажте сторінку у браузері і побачите, що у вас залишиться лише 3 студента. Також, щоб перейти на другу сторінку із студентами можете скористатись наступною адресою: “<http://localhost:8000/?page=2>”.

Якщо студентів замало, додайте їх хочаб 7-10 штук в адміністративній частині Django.

Тепер нам потрібно, щоб елементи посторінкової навігації генерували саме такі URL адреси із параметром ‘page’ всередині. А самі елементи навігації формувались динамічно в циклі згідно кількості сформованих сторінок об’єктом Paginator. Відкриваємо “students_list.html” і викидаємо статичні елементи списку залишаючи лише один, який огортаємо в цикл:

Динамізуємо посторінкову навігацію

```
1  {% if students.has_other_pages %}  
2    <nav>  
3      <ul class="pagination">  
4        <li><a href="{% url "home" %}?page=1">&laquo;</a></li>  
5        {% for p in students.paginator.page_range %}  
6          <li {% if students.number == p %}class="active"{% endif %}>  
7            <a href="{% url "home" %}?page={{ p }}">{{ p }}</a>  
8          </li>  
9        {% endfor %}  
10       <li>  
11         <a href="{% url "home" %}?page={{ students.paginator.num_pages \  
12           }}">  
13           &raquo;</a>  
14       </li>  
15     </ul>  
16   </nav>  
17   {% endif %}
```

Першим ділом ми огорнули увесь тег “nav”, контейнер нашої посторінкової навігації, в умовний оператор “has_other_pages” атрибут об’єкта Page (так, students тепер не є об’єктом типу QuerySet, а є об’єктом типу Page після того, як ми на сервері застосували Paginator пакет) повертає True, якщо, крім поточкої сторінки, існують ще інші сторінки навігації. Таким чином немає змісту показувати навігацію, якщо студентів менше, ніж 4.

В рядку 4 ми динамізували перший елемент навігації, який дозволяє користувачу повернутись на першу сторінку навігації. Тег “url” дав нам лінк на головну сторінку, а вже до нього ми додали параметр ‘page’ із значенням ‘1’. Подібним чином ми оновили елемент навігації на останню сторінку (рядок 11). Але там ми скористалися уже згаданим раніше атрибутом “num_pages” об’єкта Paginator. Сам об’єкт Paginator ми отримали звернувшись до атрибуту “paginator” об’єкта сторінки (Page, у нашому випадку це змінна під назвою students).

Також Paginator має інший корисний атрибут `page_range`¹⁹², який повертає список із номерами усіх сторінок. Таким чином пробігшись по ньому ми можемо досить легко динамічно сформувати список “li” тегів від 1 і до останнього номера сторінки. В рядку 5 ми запускаємо цикл, де поточний номер навігаційної сторінки присвоюється змінній “`r`”, яку ми далі успішно використовуємо в тілі циклу.

В стрічці 6 ми також додаємо поточній сторінці навігації клас “`active`”, що дає гарний візуальний індикатор користувачу про поточно вибрану сторінку. В тегу умови ми скористалися виразом, який використовує атрибут `number`¹⁹³ об’єкта Page. Цей атрибут повертає номер поточної сторінки навігації.

В рядку номер 7 ми використовуємо змінну “`r`” (номер сторінки навігації), щоб згенерувати правильну адресу для навігаційник сторінок. Доожної адреси додаємо параметр ‘`page`’, що рівний номеру ітерації циклу “`r`”. Також цей номер вставляємо для візуального представлення всередину тегу лінка “`a`”.

Тепер можете спробувати оновити вашу сторінку і поклікати по лінках навігації. Якщо все зробили правильно, повинні навігувати по сторінках студентів, на кожній з яких (за виключенням останньої - там вже є залишки студентів, якщо загальне число студентів не є кратним трьом) буде по 3 студента.

...

А тепер застосуйте сортування по одній із колонок, а тоді клікніть на наступну сторінку в навігації. Що зауважили?

Правильно. Сортування зникло після застосування посторінкової навігації. Це не є дуже добре. Зазвичай, якщо ми сортуємо таблицю, то хочемо переглянути наступні сторінки студентів також із застосованим напередодні порядком.

Це можна легко пофіксити просто додавши параметри ‘`order_by`’ та ‘`reverse`’ до усіх лінків, що знаходяться в навігаційному віджеті:

¹⁹²http://djbook.ru/rel1.7/topics/pagination.html#django.core.paginator.Paginator.page_range

¹⁹³<http://djbook.ru/rel1.7/topics/pagination.html#django.core.paginator.Page.number>

Робимо так, щоб навігація пам'ятала про наше сортування

```
1  {% if students.has_other_pages %}  
2  {% with order_by=request.GET.order_by reverse=request.GET.reverse %}  
3  <nav>  
4      <ul class="pagination">  
5          <li>  
6              <a href="{% url "home" %}?page=1&order_by={{ order_by }}&reverse={{ reverse }}>&laquo;</a>  
7          </li>  
8          {% for p in students.paginator.page_range %}  
9              <li {% if students.number == p %}class="active"{% endif %}>  
10                 <a href="{% url "home" %}?page={{ p }}&order_by={{ order_b\\y }}&reverse={{ reverse }}>{{ p }}</a>  
11                 </li>  
12             {% endfor %}  
13             <li>  
14                 <a href="{% url "home" %}?page={{ students.paginator.num_pages }}&order_by={{ order_by }}&reverse={{ reverse }}>  
15                     &raquo;</a>  
16                 </li>  
17             </ul>  
18         </nav>  
19     {% endwith %}  
20     {% endif %}
```

Ми огорнули нашу навігацію в тег “with”, в якому визначили змінні “order_by” та “reverse”, щоб уникати дублюючих повторів в подальшому коді. Далі до кожного посилання ми додали наші два параметри сортування, значення для яких ми взяли просто із запиту (наперед визначених змінних в тегу “with”). Так просто ми вирішили дану проблему.

Спробуйте тепер посортувати по колонці в таблиці і потім скористатись навігацією. Усе має працювати.

На домашнє завдання. Реалізуйте посторінкову навігацію повністю використовуючи Paginator пакету. Так, перепишіть всю логіку з нуля, де будете обчислювати і формувати під-список студентів, що відповідає поточній сторінці. А також прийдеся вичисляти повну кількість сторінок і тому подібні параметри.

Трохи забігаючи наперед пропоную самостійно спробувати розібратись із Javascript та технологією Ajax, і спробувати реалізувати навігацію з допомогою кнопки “Load More...” (Завантажити Більше). Думаю ви вже неодноразово бачили такий тип навігації, коли доскролюєте до кінця списку елементів, а там внизу бачите кнопку “Load More...”, при кліку по якій, динамічно довантажується наступна пачка елементів. І так продовжується до тих пір поки ви не витягнете усіх елементів із сервера. В такому випадку кнопка “Load More...” зникає.

Ще трохи ускладнюючи приклад із кнопкою “Load More...” пропоную зробити варіант, коли цієї кнопки немає, а натомість наступна пачка елементів завантажується автоматично, коли ви проскролили до кінця сторінки. І так продовжується допоки елементи на сервері не вичерпаються повністю. Тут вам допоможе Javascript подія “onscroll” та наступний вираз: `“(window.innerHeight + window.scrollY) >= document.body.offsetHeight”`. Розберіться із даним виразом самостійно. І рекомендую не писати даний функціонал на “голому” Javascript, а користуватись бібліотекою jQuery.

Модель групи

Тепер, коли список студентів виглядає дуже і дуже непогано, можемо зайнітись групами. Першим ділом розробимо клас моделі для групи, а потім повернемось до моделі студента і додамо поле групи, яке ми пропустили у попередній секції.

Згідно специфікації проекту ми маємо реалізувати два зв'язки між таблицями студентів та груп:

- студент належить до групи із таблиці груп;
- а група має призначеного старосту групи із таблиці студентів.

Перший зв'язок ми реалізуємо з допомогою поля “student_group” всередині моделі студента, а другий з допомогою поля “leader” (староста) всередині моделі групи. При чому домовимось, що поле “student_group” є обов'язковим, адже студент мусить бути в групі. А поле “leader” буде необов'язковим. Це для того, щоб не виникло так званої проблеми “курки-яйця”, коли не зможемо створити об'єкт студента, бо ще не створено жодної групи, щоб призначити в поле “student_group”, і навпаки.

Таблиця Студентів

ID	Ім'я	Прізвище	Група
1	Роман	Джексон	2
2	Джон	Ленон	1
3	Віталій	Подоба	2
4	Андрій	Іванов	4

Таблиця Груп

ID	Назва	Староста	К-сть студентів
1	МтМ-11	2	23
2	МтМ-21	1	15
3	МФК-20	3	34
4	МтМ-31	4	20

Приклад зв'язків між таблицями в реляційній базі даних

Основна новизна цієї секції буде в освоєнні зв'язків в реляційній базі та Django ORM системі. Тому, перед тим як переходити напряму до реалізації класу моделі групи, розберемо трохи теорії.

Теорія зв'язків між таблицями

Є як мінімум три типи зв'язків між рядками таблиць в реляційній базі даних. Давайте на прикладах розглянемо кожен із них.

Один до Одного. Уявимо, що маємо базу клієнтів, а в окремій табличці тримаємо список даних по їхніх паспортах. Людина може мати лише один паспорт. Таким чином на один рядок в таблиці із паспортами може зсилатись не більше, ніж один рядок із таблиці клієнтів. І навпаки, один паспорт може належати не більше, ніж одній людині.

Цей тип зв'язку будемо називали “один до одного” (англ. One To One). I в Django ORM такий тип зв'язку позначається полем `OneToOneField`¹⁹⁴.

Багато до Одного та Один до Багатьох. У нашому випадку із базою студентів ми маємо студента, який може належати лише до однієї групи. Адже студент навчального закладу не може одразу навчатись в двох і більше групах, це нонсенс. Проте кожна із груп має багато студентів призначених.

Отже, отримуємо із сторони студента з'язок лише на один рядок в таблиці груп, проте більше, ніж один студент може зсилатись на одну і ту ж саму групу. Таким чином створюючи зворотній зв'язок однієї групи із багатьма студентами.

Цей тип зв'язку називаємо “Один до Багатьох” (або “Багато до одного”, взалежності від того, з яких сторін поставити студента та групу). В Django моделях даний тип зв'язку представлений полем `ForeignKey`¹⁹⁵. Цей тип зв'язку є одним із найчастіше використовуваних.

Багато до Багатьох. Уявіть, що маємо блог, де багато різних авторів регулярно дописує статті. При чому час до часу вони між собою кооперуються і працюють над однією статтею і публікують під спільним авторством.

¹⁹⁴<http://djbook.ru/rel1.7/ref/models/fields.html#onetoonefield>

¹⁹⁵<http://djbook.ru/rel1.7/ref/models/fields.html#django.db.models.ForeignKey>

Таким чином на даному блозі один автор зазвичай є власником більше, ніж однієї статті. А стаття, в свою чергу, інколи може мати більше, ніж одного автора - список авторів. В даному випадку маємо більше одного елемента у зв'язку “автор - стаття”.

Цей тип зв'язку називаємо “Багато до Багатьох”. В Django для нього існує поле моделей [ManyToManyField¹⁹⁶](#).

Сам на себе. Якщо клієнт привів нам іншого клієнта, тоді ми це позначаємо в таблиці як зв'язок рядка із іншим рядком цієї ж таблиці. Такий тип зв'язку називаємо ще посиланням на себе. Тобто таблиця містить рядок, що посилається на рядок з цієї ж таблиці.

В Django ORM для цього використовуємо також поле ForeignKey, але першим аргументом, замість назви класу моделі, передаємо стрічку ‘self’. Ця стрічка повідомляє ORM систему, що дане поле буде посилатись на рядки всередині тієї самої таблиці (на об’екти моделі, в якій міститься дане поле).

Кожного разу, коли ми використовуватимемо один із типів полів зв'язку між таблицями, Django для нас створюватиме в базі поле з назвою “[field_name]_id”, де зберігатиме значення первинного ключа рядка тої таблиці, на яку зсилається дане поле зв'язку.

В мові SQL зв'язок між таблицями зазвичай декларується ключовим словом FOREIGN KEY, який, в свою чергу, зазвичай зсилається на унікальне поле (PRIMARY KEY) рядка іншої таблиці. Приклад декларації FOREIGN KEY (з англ. FOREIGN - зовнішній):

¹⁹⁶<http://djbook.ru/rel1.7/ref/models/fields.html#manytomanyfield>

Створюємо таблицю із зовнішнім ключем leader

```
1 CREATE TABLE groups (
2     id integer AUTO_INCREMENT NOT NULL PRIMARY KEY,
3     title VARCHAR(256) NOT NULL,
4     leader_id integer,
5     FOREIGN KEY (leader_id) REFERENCES students(id)
6 );
```

Поле “leader_id” є зовнішнім ключем, що містить ID студента із таблиці студентів. Ключове слово FOREIGN KEY, що йде після списку полів, декларує поле “leader_id” як посилання на поле “id” в таблиці студентів. Важливо знати, що поля, які зв’язані між собою такими ключами, обов’язково повинні мати одинаковий тип. Наприклад, не може бути “leader_id” INT, а “id” DOUBLE.

Якщо зовсім просто, то зв’язок між рядками таблиць організовується через запам’ятовування в одному із полів рядка значення первинного ключа іншого рядка. Якщо на один рядок посилається більше, ніж один інший рядок, тоді автоматично маємо різні типи зв’язків відповідно до кількості посилань (один до одного, багато до одного, багато до багатьох).

Клас моделі Групи

Тепер, коли ми поверхнево пробіглись по типах зв’язків, час братись до написання класу моделі групи. Більшість речей будуть вам зрозумілі, оскільки вони подібні до визначення класу моделі студента. Єдиний новий тип поля - це буде поле зв’язку OneToOneField.

Отже, відкриваємо модуль models.py і додаємо клас моделі групи. Я зазвичай копіюю уже визначений клас студента і підправляю його для потреб групи. Це пришвидшить процес і зменшить кількість помилок:

Модель групи в models.py

```
1 class Group(models.Model):
2     """Group Model"""
3
4     class Meta(object):
5         verbose_name = u"Група"
6         verbose_name_plural = u"Групи"
7
8         title = models.CharField(
9             max_length=256,
10            blank=False,
11            verbose_name=u"Назва")
12
13        leader = models.OneToOneField('Student',
14            verbose_name=u"Староста",
15            blank=True,
16            null=True,
17            on_delete=models.SET_NULL)
18
19        notes = models.TextField(
20            blank=True,
21            verbose_name=u"Додаткові нотатки")
22
23    def __unicode__(self):
24        if self.leader:
25            return u"%s (%s %s)" % (self.title, self.leader.first_na\
26 me,
27                           self.leader.last_name)
28        else:
29            return u"%s" % (self.title,)
```

Ми назвали клас моделі Group і унаслідували його знову від базового класу Model. Також визначили подібним чином внутрішній клас Meta, щоб моделі групи відображались із підписами українською мовою. Після цього визначили

поля “title”, “leader” і “notes”. Якщо з полями “title” та “notes” все зрозуміло, оскільки ми їх розбирали при реалізації моделі студента, то поле “leader” потребує трохи детальнішого пояснення.

Одна група може мати не більше, ніж одного старосту. В свою чергу один студент не може бути старостою більше, ніж однією групи одночасно. Тобто ми маємо зв’язок типу “Один до Одного”. Саме тому скористались полем OneToOneField для поля “leader”.

Першим аргументом усіх полів зв’язку йде модель, на яку вони будуть посилятись. Це може бути стрічка з назвою класу моделі (напр. ‘Student’), або пряме посилання на клас. Таким чином ми передали першим параметром полю “leader” назву моделі студента: “Student”. Також ми вказали, що дане поле є обов’язковим та може бути порожнім на рівні бази даних (null=“True”).

В рядочку 17 є новий для нас параметр `on_delete`¹⁹⁷. Він вказує на поведінку бази даних у випадку, якщо із таблиці студентів буде видалений студент, на який зсилається дана група. Є кілька можливих варіантів поведінки. Ми обрали “SET_NULL”. Це означає, що групі автоматично встановиться порожнє значення в полі “leader”, якщо студента видалять із таблиці студентів. Іншими варіантами може бути заборона видалення студента, на якого зсилається група, видалення також і групи, а також встановлення дефолтного значення поля (якщо воно є у визначеній поля).

Поле OneToOneField зробить неможливим призначення одного і того ж студента більше, ніж одній групі в Django адмінці. Таким чином усі налаштування полів моделей логічно відображаються в адміністративній частині фреймворка.

На завершення розбору класу моделі групи маємо ще звернути увагу на метод “`__unicode__`”, який є дещо складнішим, ніж у класі моделі студента. В Django адмінці об’єкти групи відображатимуться у вигляді: “Назва Групи (Ім’я Прізвище)”, де Ім’я та Прізвище будуть полями старости групи. Але оскільки поле старости не є обов’язковим, тому маємо передбачити варіант,

¹⁹⁷ http://djbook.ru/rel1.7/ref/models/fields.html#django.db.models.ForeignKey.on_delete

коли воно не встановлене. Тоді ми просто відображатимемо назву групи. Саме для цього ми і скористалися умовним оператором з умовою “if self.leader”, щоб перевірити чи староста групи вже є.

Щоб скористатись моделлю групи потрібно її спочатку активувати. Зробимо це так само як ми проробляли із моделлю студента. Скористаємося командами створення та запуску міграційного модуля:

Створюємо міграцію

```
1 # створюємо модуль для міграції
2 (studentsdb)$ python manage.py makemigrations students
3
4 # а тепер запускаємо міграцію; нам не потрібно вказувати
5 # яку саме міграцію запускати; Django для нас запам'ятує
6 # та запускає лише ті міграції, які ще не були запущені на
7 # нашій базі даних
8 (studentsdb)$ python manage.py migrate
```

Після вище наведених команд матимемо в базі нову таблицю “students_group” із колонками полів моделі групи. Тепер давайте трохи поекспериментуємо із моделлю групи та студента. Особливо цікавим для нас є поле зв’язку групи із її старостою:

Освоюємося із полями зв’язку в shell

```
1 (studentsdb)$ python manage.py shell
2 Python 2.7.8 (default, Oct 31 2014, 21:10:17)
3 [GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
4 Type "help", "copyright", "credits" or "license" for more information
5
6 (InteractiveConsole)
7 >>> from students.models import Student, Group
8
9 # беремо першого студента з таблиці студентів
10 >>> stud1 = Student.objects.all()[0]
11 >>> stud1
```

```
12 <Student: Oleg Podoba>
13
14 # створюємо групу і призначаємо її старостою студента stud1
15 >>> group1 = Group(title="МТМ", leader=stud1)
16
17 # пробуємо зберегти новостворену групу в базу
18 >>> group1.save()
19 Traceback (most recent call last):
20   File "<console>", line 1, in <module>
21     File "/data/work/virtualenvs/studentsdb/lib/python2.7/site-packages\
22 s/django/db/models/base.py", line 591, in save
23       force_update=force_update, update_fields=update_fields)
24
25 ...
26
27   File "/data/work/virtualenvs/studentsdb/lib/python2.7/site-packages\
28 s/MySQLdb/connections.py", line 36, in defaulterrorhandler
29     raise errorclass, errorvalue
30 IntegrityError: (1062, "Duplicate entry '1' for key 'leader_id'")
31
32 # помилка сказала нам, що даний студент вже є старостою іншої
33 # групи; у вас не виникне дана помилка, адже у вас в базі
34 # ще немає інших груп;
35 # для того, щоб повторити дану помилку потрібно спробувати
36 # також створити ще одну групу і встановити того ж студента
37 # у ній старостою;
38
39 # у такому випадку я спочатку відв'яжу студента від
40 # попередньо прив'язаної групи, а потім знову спробую
41 # перезберегти новостворену групу;
42
43 # ось ця група винуватиця; я знайшов її фільтруючи по
44 # полю leader_id="1", адже ми витягнули первого студента
45 # з таблиці;
46 >>> group2 = Group.objects.filter(leader_id='1')
```

```
47 >>> group2
48 [<Group: demo 1 (Oleg Podoba)>]
49
50 # встановимо значення поля leader в порожнє значення
51 # і збережемо цю групу:
52 >>> group2.leader = None
53 >>> group2.save()
54
55 # тепер без проблем зможемо зберегти в базу group1:
56 >>> group1.save()
57 >>> Group.objects.filter(title="MTM")[0]
58 <Group: MTM (Oleg Podoba)>
59
60 # а тепер фокус:
61 >>> stud1.group
62 <Group: MTM (Oleg Podoba)>
63
64 # як бачите в об'єкта студента з'явився атрибут group,
65 # хоча ми його ніде ще не згадували; це для нас зробив
66 # Django ORM і вклав туди об'єкт групи, для якої stud1 є
67 # старостою;
68
69 # якщо stud1 не буде старостою групи, тоді і атрибут
70 # group запрацює по-іншому:
71 >>> group1.leader = None
72 >>> group1.save()
73 >>> stud1.group
74 Traceback (most recent call last):
75   File "<console>", line 1, in <module>
76     File "/data/work/virtualenvs/studentsdb/lib/python2.7/site-packages\
77     s/django/db/models/fields/related.py", line 428, in __get__
78       self.related.get_accessor_name()
79 RelatedObjectDoesNotExist: Student has no group.
80
81 # виходимо з допомогою клавіш Ctrl-D
```

До даної сесії “shell” варто додати, що кожне поле зв’язку в Django ORM створює додаткові атрибути також і на об’єктах моделі, на яку посилається те чи інше поле зв’язку. Таким чином не лише група знає, якого їй старосту призначили, але й студент знає, що він являється старостою групи. Це ми називаємо зворотніми автоматичними зв’язками.

Для полів OneToOneField в об’єкті моделі, на яку посилаються, створюється атрибут із назвою класу моделі, яка посилається на нього. Таким чином “leader” поле в моделі групи також створює поле “group” в об’єкті моделі студента. Дане поле “group” міститиме об’єкт групи, для якої даний студент є старостою.

У випадку, якщо використовуємо ForeignKey або ManyToManyField поля, тоді в дію вступає об’єкт класу [RelationManager](#)¹⁹⁸ з цілім набором методів для роботи із зв’язаними об’єктами. Він з легкістю дозволяє не лише шукати зв’язані об’єкти, але й повністю керувати набором таких зв’язків на об’єкти моделі.

Тут¹⁹⁹ можете детальніше ознайомитися із усіма можливостями полів зв’язку в Django ORM.

Тепер, коли маємо завершену модель для групи можемо оновити нашу сторінку із списком груп та реалізувати у в’юшці роботу із базою даних. Це вам на домашнє завдання. Таким самим чином, як ми це зробили для списку студентів, реалізуйте список груп із бази даних, його сортування та посторінкову навігацію.

Оновлюємо модель студента

У нас ще залишився один борг в моделі студента. Нам потрібно додати поле групи. Але назвати його іменем “group” ми не можемо, адже буде конфлікт із зворотнім автоматичним зв’язком, який приходить в модель студента завдяки полю “leader” в моделі групи.

¹⁹⁸ <http://djbook.ru/rel1.7/ref/models/relations.html>

¹⁹⁹ <http://djbook.ru/rel1.7/ref/models/relations.html>

Загалом маємо ще 2 варіанти, як обійти цю проблему із назвою поля. Можна полю “leader” в моделі групи дати значення “+” додатковому параметру “related_name” і тоді зворотній зв’язок ми відключимо. Але це ще може нам пригодитись. Іншим варіантом є встановлення будь-якої іншої стрічки-імені у значення аргумента “related_name”. В такому випадку Django ORM використовуватиме не “group” для атрибута в об’єкті студента, а ту стрічку, яку ми передамо полю в якості параметра “related_name”.

Ми підемо шляхом найменшого опору і просто назовемо поле “student_group”. Данем поле бути зв’язком типу багато до одного, адже студентів є багато в одній групі, а один студент не може бути старостою більше, ніж в одній групі. Тому використовуємо поле ForeignKey:

Додаємо поле student_group всередину класу моделі студента, models.py

```
1 student_group = models.ForeignKey('Group',
2                                 verbose_name=u"Група",
3                                 blank=False,
4                                 null=True,
5                                 on_delete=models.PROTECT)
```

Зверніть увагу на те, що ми встановили іншу поведінку полю student_group при видаленні групи, на яку вказує дане поле. В даному випадку ми використали режим “PROTECT” в параметрі “on_delete”. Це означає, що нам не дозволять видалити групу допоки існує хоча б один студент, що її використовує в своєму полі “student_group”. Спробуйте поекспериментувати в Django адмінці і самі переконаєтесь в цьому.

Щоб нове поле з’явилось також в базі даних в таблиці студентів, потрібно знову запустити міграційні скрипти:

```
1 (studentsdb)$ python manage.py makemigrations students
2 (studentsdb)$ python manage.py migrate
```

Тепер можете зайти в адмінку і поекспериментувати з полем старости групи в моделі групи та полем групи в моделі студента.

На домашнє завдання: запустіть “shell” та поекспериментуйте із новододаним полем “student_group” в клас моделі студента. Спробуйте створити 2 різних студента і додати їх до новоствореної групи. Зверніть увагу на атрибут “student_set” в об’єктах груп.

В подальших главах ми будемо з вами активно використовувати дані зв’язки для фільтрування студентів по групах, а також на наших власних формах управління студентами та групами.

Фікстури та міграції

На завершення даної глави розглянемо поняття фіктур, початкових даних проекта та промігруємо із sqlite на MySQL базу даних.

Фікстури

В процесі розробки сторінки із списком студентів ми з вами частенько відвідували адмін частину Django і спеціально для тестування нашого коду створювали демо студентів і групи.

Так от. Для того, щоб не потрібно було кожного разу руками переклікувати Django адмінку і створювати набір демо контенту, існує інший спосіб набивки даних. Цей спосіб використовується для:

- підготовки даних для вашого власного робочого середовища;
- для допомоги членам вашої розробницької команди отримати ту ж саму конфігурацію даних для розробки;
- інколи навіть для підготовки початкового контенту на продакшин сайтах.

Сам підхід є надзвичайно простий. Готуєте файлик у потрібному форматі. Йх є декілька, але ми будемо користуватись саме форматом даних **JSON²⁰⁰**. В ньому

²⁰⁰<http://uk.wikipedia.org/wiki/JSON>

ми описуємо список об'єктів (рядків в таблицях бази даних), даних та назв моделей, що їх використовують. Ось приклад такого файла:

Приклад JSON фікстури

```
1  [
2  {
3      "model": "students.student",
4      "pk": 1,
5      "fields": {
6          "first_name": "Vitaliy",
7          "last_name": "Podoba",
8          "photo": "./podoba3.jpg",
9          "student_group": 1,
10         "birthday": "1984-17-06",
11         "ticket": "911"
12     }
13 },
14 {
15     "model": "students.group",
16     "pk": 1,
17     "fields": {
18         "title": "MTM",
19         "leader": 1,
20         "notes": ""
21     }
22 }
23 ]
```

Такий файл якраз і називається фікстурою (fixture, з англ. заготовка, арматура). Наперед заготований набір даних для вашої бази даних.

Після того як подібний файл є підготований у вашій файловій системі, його можна “згодувати” команді “`loaddata`” скрипта `manage.py`. Але ми цього робити не будемо, адже дані у нас в базі уже є. Натомість розберемось як не створювати даного файла вручну, адже редактування тексту це завдяки трудоємко. Повірте, ця книга дала мені це дуже добре зрозуміті.

Зазвичай швидше перший раз набити дані в базі через адмінку, а тоді їх залинути в такий файл і усі наступні рази його використовувати для отримання бази з демо даними.

Ось як скопіювати усі дані бази в такий файл:

Використовуємо команду `datadump`, щоб отримати наші дані з бази у форматі JSON

```
1 (studentsdb)$ python manage.py dumpdata --format=json --indent=4 > d\
2 demo_data.json
```

Таким чином отримаємо файл із назвою `demo_data.json` в поточній директорії. Параметром “`indent`” ми вказали, що хочемо мати гарно форматований файл із 4-ма відступами між вкладеними рівнями даних в JSON форматі.

Кожного разу, коли ви наламали дров у своїй базі, або просто хочете знову почати із первинний даних, повністю очищаете базу даних командою “`flush`”, а тоді перезбурите базу з нуля з допомогою щойно підготовленого файлка `demo_data.json`.

Увага, перед виконанням наступних команд зробіть резервну копію вашого sqlite файла бази даних. У випадку, якщо щось піде не так, ви матимете бекап.

Перезбуриємо дані бази з нуля

```
1 # видаляємо усі дані з бази
2 (studentsdb)$ python manage.py flush
3
4 # а тепер відновлюємо дані бази;
5 (studentsdb)$ python manage.py loaddata --app=students demo_data.json
```

Якщо ж потрібно створити базу даних з нуля разом із самою структурою бази і таблиць, тоді перед використанням команди “`loaddata`” потрібно також скористатись командою “`migrate`”.

Детальніше про створення початкових даних можете ознайомитись [тут²⁰¹](#).

Міграція на MySQL

Тепер, коли ми вміємо витягнути і встановити назад набір наших студентів і груп (та й взагалі усіх інших даних бази) в базі, можемо досить легко промігрувати із бази даних sqlite на MySQL.

База даних sqlite є простою у використанні та встановленні, але у ній бракує багато функціоналу, що є досить важливим для великої частини продакшін веб-аплікацій. Тому в цій секції переїдемо на MySQL і далі над проектом працюватимемо використовуючи дану базу даних.

Якщо ви виконали домашні завдання глави книги “Робоче Середовище”, тоді у вас уже має бути встановлена база даних MySQL разом із необхідною базою для нашої аплікації. База даних під назвою “students_db” та користувач “students_db_user” уже мають бути готові для нашого використання. Також пригадайте пароль, який ви дали користувачу “students_db_user”. Він нам тут пригодиться.

Ми уже підготували файл `demo_data.json` із усіма поточними даними в базі. Тому все, що залишається це переключити налаштування проекту на нову базу даних та закинути дані в базу командою “`loaddata`”.

Переконайтесь, що ваш MySQL сервер запущений. Як керувати MySQL сервером ми обговорювали в главі “Робоче Середовище”, тому при потребі зверніться до неї.

Відкриваємо модуль налаштувань проекту `settings.py` і оновлюємо змінну `DATABASES`:

²⁰¹<http://djbook.ru/rel1.7/howto/initial-data.html>

```
1 DATABASES = {  
2     'default': {  
3         'ENGINE': 'django.db.backends.mysql',  
4         'HOST': 'localhost',  
5         'USER': 'students_db_user',  
6         'PASSWORD': '*****',  
7         'NAME': 'students_db',  
8     }  
9 }
```

Ми оновили ключ ‘default’ в змінній-словнику **DATABASES**²⁰². Ми переключили ‘ENGINE’ з sqlite на mysql двигунець, який дозволить нам спілкуватись саме із MySQL базою даних. MySQL комунікатор вимагає дець інших параметрів:

- ‘HOST’: адреса сервера, на якому запущений наш MySQL сервер;
- ‘PORT’: порт, на якому запущений наш MySQL сервер; ми не передали даний ключ, адже наш MySQL сервер працює на дефолтному порті (3306);
- ‘NAME’: база даних нашого проекту;
- ‘USER’: користувач, який має доступ до даної бази;
- ‘PASSWORD’: пароль користувача (введіть свій коректний пароль замість зірочок).

Після цього ми готові до створення структури даних та імпорту даних в базу:

Створюємо структуру бази та набиваємо дані

```
1 (studentsdb)$ python manage.py migrate  
2 (studentsdb)$ python manage.py loaddata demo_data.json
```

²⁰²<https://docs.djangoproject.com/en/1.7/ref/settings/#databases>

В даному випадку ми імпортнули в базу не лише дані по аплікації ‘students’, але й усі дані. Адже ми з нуля створювали структуру і усі дані в MySQL базі.

Якщо останні команди закінчились без помилок, тоді можна запускати сервер і переконатись, що ваша аплікація працює один до одного як працювала з базою sqlite.

Хочете ще практики з базами даними і почуватись як риба у воді налаштовуючи їх? Зайнсталуйте та налаштуйте базу даних PostgreSQL. Промігруйте дані на неї та запустіть ваш Django проект на PostgreSQL. Прийдеться також навчити ваш Python говорити із PostgreSQL базою.

db.py

На завершення глави рекомендую винести змінну DATABASES за межі модуля settings.py. Наприклад в модуль db.py, який ми не будемо додавати в репозиторій. Після переключення на базу MySQL ми вписали пароль доступу до бази, тому тримати такі дані в репозиторії є погано (особливо якщо він у вас публічний).

На домашнє завдання попрошу вас зробити це самостійно. А потім звірити свій результат із кодом, що йде разом із книгою.

Домашнє завдання

Вітаю вас! Ви пройшли ще одну дуже важливу главу книги. І тепер у вас є розуміння, що таке дані, структури даних, як зберігаються дані в базі.

Ми з вами оглянули способи роботи з даними всередині Django і в процесі даної глави реалізували:

- моделі студента та групи;
- сортування на навігація по списку студентів;

- міграцію на базу MySQL.

Гарний шмат роботи! Будь-ласка звертайтесь до цієї глави регулярно, коли бракуватиме знань по роботі із базою даних та моделями. Думаю не все може бути зрозумілим з першого разу. В такому випадку переходить до практики, ковиряйтесь в коді, потім знову вертайтеся до теорії. Через кілька таких кіл ви все більше і більше розумітимете матеріал даної глави.

...

Закомітьте усі зміни в репозиторій. А краще робити це після кожної секції в главі.

Для закріплення матеріалу даної глави проробіть усі домашки, які ви знайшли всередині секцій, а також:

- організуйте моделі в підпакети так само як ми це зробили із в'юшками; наприклад в папці аплікації створіть пакет “models”, в який покладіть модулі `students.py` та `groups.py`, що міститимуть моделі студента та групи відповідно;
- повністю реалізуйте сторінку списку груп самостійно; все аналогічно до того, як ми це зробили із списком студентів; вигляд списку студентів шукайте в главі “Специфікації проекту”;
- нагадую про посторінкову навігацію “Load more...” та з нуля без використання пакету “paginator”;
- спробуйте придумати та реалізувати модель для журналу відвідування.

Об’ємніше завдання: реалізувати закладку Іспити. Для цього необхідний мінімум речей: модель іспита та в’юшка із списком іспитів. Модель іспиту містить поля:

- назва предмету;
- дата і час проведення;
- назва викладача, що прийматиме іспит;

- група, для якої проводитиметься іспит.

Список іспитів сформуйте в подібному стилі як ми маємо список студентів. А щодо полів та їх сортування - на ваш розсуд. Зробіть так, щоб вам самим подобалось та було зручно в користуванні. Для тих, кому це вдастся менше, ніж за 4 години, додаткове завдання: реалізувати також результати іспитів. Кожен результат прив'язується до іспиту та студента полями зв'язку і звісно має оцінку. Користувацький інтерфейс і структура URL адрес тут буде дещо складніша.

Ще трохи завдань, щоб надихнути вас на читання чужого коду:

Для супер-ніндзя: відвідайте пакет “django.db”, знайдіть код методу save (той метод, яким ми часто користувались протягом цієї глави, щоб зберігати об’єкти в базу) та спробуйте розібратись як він працює. Чесно признаюсь, я пробував, але з першого разу далеко не все зміг зрозуміти. Якщо вам вийде все розібрати в ньому - пишіть в групу підтримки пояснення, я повчусь ;-)

Для тих, хто ще не ніндзя, рекомендую відвідати вбудовану в Django аплікацію “statusmessages” і спробувати розібратись як вона працює базуючись на її внутрішньому коді. Час починати читати чужий код, зрозуміти його і вчитись. Спочатку буде все складно і не зрозуміло, але згодом ситуація покращуватиметься. Вміння читати чужий, якісний код - це один із показників вашого рівня як програміста.

...

Наступна глава буде третьою базовою главою даної книги. Таким чином, якщо ви успішно освоїте 6-у, 7-у та 8-му глави книги, Django практично у вас в кишенні.

Побудова URL структури сайту, в’юшки, шаблони, робота з базою даних, робота з формами - це основа веб-розробки з Django. Тому, допоки ви не освоїти успішно дані три глави, будь-ласка, не рухайтесь далі по книзі. Все інше - це другорядні речі, які ви все одно легко зрозумієте після розбору основ. Якщо ви зареєстровані в закритій групі підтримки, тоді чекаю ваших домашніх завдань. Поставте собі за ціль виконати хоча б половину з них, а вже тоді рухатись далі до наступної глави, у якій ми розберемось із веб-формами та формами в Django.

8. Форми роботи із студентом та групою: Django форми, валідація

Ось і дійшли ми з вами до глави, в якій розберемось із формами.

Веб-форми - це інструмент, з допомогою якого користувач може відправити дані на сервер, а не лише використовувати свій браузер у режимі отримування даних. Форми - це дуже показовий елемент веб-розробки, що свідчить про рівень програміста. З однієї сторони це доволі рутинна і трудоємка робота, а з іншої сторони, створити по-справжньому зручні і прості в користуванні форми, є далеко непростою задачею.

Тому дана глава містить доволі важливий і складний матеріал. А саме:

- верстка HTML форми;
- обробка форми на сервері, валідація даних;
- використання вбудованих класів Django форм, форм моделей;
- форми з Twitter Bootstrap та аплікацією Django Crispy Forms;
- кастомізація Django адмінки.

В процесі даної глави ми з вами:

- реалізуємо форму додавання студента практично з нуля;
- побудуємо форму редагування студента використовуючи Django форми моделей;
- реалізуємо функцію видалення існуючих студентів;
- створимо форму контактування адміністратора на закладці Контакт;
- покращимо список студентів в адміністративній частині та додишемо валідатори для поля “Група” на формі редагування студента.

Наприкінці у вас буде повний арсенал роботи із веб-формами як на клієнтській стороні, так і на стороні сервера.

А почнемо ми із невеликої кількості теорії, а саме - HTML форми.

HTML Форми

Ми вже з вами зовсім коротко захопили теги форм і полів у главі із статичною версткою веб-сторінки. Тут же ж трохи детальніше глянемо на необхідний мінімум для ефективної побудови правильних форм на сайті.

Огляд

Для користувача форма повинна бути легка, зручна та швидка у використанні. Будь-яка дія користувача, в ідеалі, повинна мати миттєвий фідбек зі сторони нашого коду. Якщо дані введені коректно, тоді після відправки даних на сервер, користувач повинен бачити повідомлення, що його запит успішно оброблений, а поля форми обнулені (знову порожні). Якщо ж дані були введені некоректно, тоді форма зберігає попередньо введені користувачем дані в полях, а також відображає додаткові повідомлення конкретизуючи помилки користувача для подальшого виправлення.

Також, якщо форма була відправлена і оброблена коректно, варто редіректити користувача на іншу сторінку, де повідомляти його про успішність операції.

На серверній частині є дуже важливим не просто отримувати та зберігати дані від користувача, а кожного разу перевіряти ввід даних на коректність. Інакше можемо опинитись із некоректними даними в базі, або, що ще гірше, поламаною базою даних та “хакнутими” веб-сторінками.

Поля форми та вигляд самої форми варто оформляти у зручному для користувача вигляді, додаючи необхідну кількість пояснень, щоб вберегти його від неправильно введених даних.

Для підвищення ефективності роботи із формою, в наш час, велику частину полів динамізують та покращують з допомогою багатого Javascript функціоналу. Наприклад, щоб вводити дані щодо дати і часу, використовується віджет календаря. Для вибору кольору, використовуються так звані “Color Picker”.

HTML код форми

Головний тег при роботі із формами є, звісно, “form”. Це невидимий на сторінці тег, який вказує на властивості форми, а також містить поля та кнопки цієї форми.

Найважливішими атрибутами тегу форми є:

- **action**: вказує адресу, на яку відправляти дані форми;
- **enctype²⁰³**: тип кодування даних форми перед відправкою на сервер (може бути одним із трьох: “text/plain”, “multipart/form-data” та “application/x-www-form-urlencoded”; просто запам’ятайте, що потрібно використовувати значення “multipart/form-data”, якщо маєте поля файлів на формі);
- **method**: метод запиту форми (може бути GET або POST); більше про цей атрибут у наступних секціях;
- **name**: назва форми; форм на сторінці може бути багато і в такому випадку кожній із них варто надавати інше ім’я; з допомогою імені можна на сервері визначати, яка саме форма була відправлена (особливо, якщо форми мають одинаковий “action” атрибут);
- **target²⁰⁴**: вказує, де саме показувати результат відправки форми.

Ось приклад порожньої форми із визначеними необхідними атрибутами:

Тег form

```

1 <form action="/submit_form.html" method="post"
2     enctype="multipart/form-data" name="myform">
3 </form>

```

Поки дана форма без полів і кнопок. Всередину тегу form можемо додавати один або більше наступних тегів:

²⁰³<http://htmlbook.ru/html/form/enctype>

²⁰⁴<http://htmlbook.ru/html/form/target>

- **input²⁰⁵**: найпоширеніший тег всередині форм; може мати кілька різних типів взалежності від атрибути “**type**”²⁰⁶ (checkbox, radio, text, hidden, submit і ще багато інших);
- **textarea²⁰⁷**: дозволяє вводити користувачу не лише стрічку тексту, а цілі параграфи тексту;
- **select²⁰⁸**: набір опцій для вибору доступних у так званій “випадайці” (випадаюче меню); працює у двох режимах дозволяючи вводити як лише одне значення, так і кілька значень одночасно (атрибут “multiple”); опції для вибору формуються з допомогою вкладених тегів “**option**”²⁰⁹;
- **button²¹⁰** або **input[type=submit]**: кнопки на формі; в основному використовуються для відправки форми на сервер;
- **label²¹¹**: підписує поле поясннювальним текстом; за допомогою атрибути “for” можна фокусувати поле при кліку по тегу “label”.

Знаючи та розуміючи, коли саме використовувати вищеперечислені теги і їхні атрибути, ви без проблем зможете реалізувати 90% необхідних у вашому проекті форм.

button vs input[type=submit]: є два теги, з допомогою яких можна реалізувати кнопки форми. Обидва є рівноцінними по функціоналу. Але button дає кращі візуальні можливості. Зокрема, в тег button можна вкладати внутрішній контент. Тому, в принципі, не надто важливо яким саме тегом ви будете користуватись.

А тепер, давайте зверстаемо просту форму, де скористаємося кількома із вищезгаданих тегів:

²⁰⁵<http://htmlbook.ru/html/input>

²⁰⁶<http://htmlbook.ru/html/input/type>

²⁰⁷<http://htmlbook.ru/html/textarea>

²⁰⁸<http://htmlbook.ru/html/select>

²⁰⁹<http://htmlbook.ru/html/option>

²¹⁰<http://htmlbook.ru/html/button>

²¹¹<http://htmlbook.ru/html/label>

Демо HTML форма

```
1 <form action="/submit_form.html" method="post"
2     enctype="multipart/form-data" name="myform">
3
4     <input type="hidden" value="27" name="student_id" />
5
6     <div>
7         <label for="first_name">Ваше ім'я</label>
8         <input type="text" value="" name="first_name"
9             id="first_name" />
10    </div>
11
12    <div>
13        <label for="graduated">Закінчили навчання?</label>
14        <input type="checkbox" checked="1" name="graduated"
15            value="1" id="graduated" />
16    </div>
17
18    <div>
19        <label for="gender">Ваша стать</label>
20        <input type="radio" name="gender" value="male"
21            id="gender" />
22        <input type="radio" name="gender" value="female" />
23    </div>
24
25    <div>
26        <label for="age">Ваш вік</label>
27        <select name="age" id="age">
28            <option value="">
29                Будь-ласка, виберіть ваш вік
30            </option>
31            <option value="10">Більше ніж 10</option>
32            <option value="20">Більше ніж 20</option>
33        </select>
34    </div>
```

35

```
36 <input type="submit" name="save_button"
37     value="Зберегти" />
38
39 </form>
```

Якщо замість кирилиці в браузері отримали “абру-кадабру”, тоді додайте шапку документа (тег head) і тег із кодуванням UTF-8: ‘`<meta charset="UTF-8"/>`’. Після цього потрібно перезавантажити сторінку.

Як бачите, тег форми може містити і інші теги. Ми огорнули кожне поле в тег “div”, щоб логічно погрупувати поле вводу та відповідну мітку (тег “label”). Крім того, даний тег гарно поскладав кожне поле на окрему стрічку.

Зверніть особливу увагу на те, як працюють поля “checkbox”, “radio” і “select”.

Ось як виглядатиме дана форма в браузері:

Демо Форма

Ваше ім'я

Закінчили навчання?

Ваша стать

Ваш вік

Демо форма

На домашнє завдання: розширте дану форму більшою кількістю полів так, щоб усі вищеперечислені теги форми були задіяні.

Цього нам буде достатньо, щоб рухатись далі до практики. Але перед практикою розглянемо ще кілька цікавих моментів.

GET vs POST

Веб-форми можуть робити запити на сервер двох типів: GET і POST.

Як ми уже пам'ятаємо із попередніх глав, коли ми навігуємо в інтернеті, відкриваючи сторінки з допомогою URL адреси браузера, а також клікаючи по лінках, ми автоматично робимо запити на сервер типу GET. Цей тип запиту отримує дані із сервера.

Для відправки даних на сервер у формі частіше використовується метод POST. Він дозволяє відправити більшу кількість даних, в той час як метод GET має дуже обмежений об'єм даних для відправки на сервер.

Якщо вам потрібно зробити форму, яка змінює дані на сервері, тоді використовуйте метод запиту POST. Якщо ж лише отримати дані із сервера і при цьому об'єм даних відправляється невеликий, тоді метод GET.

Також метод POST кращий тоді, коли форма містить поля, які не варто відображати в URL адресі браузера. Наприклад пароль. З іншої сторони, якщо адреса запиту із усіма полями потрібна для подальшого запам'ятовування користувачу (наприклад в закладці), тоді варто скористатись методом GET. Це буває корисно для форм пошуку та фільтрів даних, щоб пізніше користувач міг легко потрапити на попередньо відконфігурковану сторінку із результатами пошуку.

Таким чином POST буде підходящим для форм додавання, редагування та видалення студента. В той час, як GET краще підійде для форми пошуку на сайті.

Валідація

Якщо занадто довіряти усім даним, що приходять до нас від користувача, то можемо закінчити повністю поламаними базою даних та веб-сайтом загалом.

Щоб уникати подібних ситуацій, зазвичай, дані форм перевіряються на коректність. В ідеалі дану перевірку роблять у двох місцях: на стороні браузера та на стороні клієнта.

Якщо перевірка на веб-сторінці є необов'язковою, і швидше служить для зручності користувача та швидкості роботи із формою, то серверна валідація

даних - це просто необхідність.

Найбільш поширеними прикладами валідації форм є:

- перевірка на введення даних у обов'язкові поля форми;
- перевірка на правильний формат телефонів, емейлів, дат;
- перевірка на достатньо складний та правильний ввід пароля;
- правильний тип файлу та обмеження на розмір файлу.

Наше завдання, як розробників, на сервері, перед виконанням будь-яких операцій над даними із форми, перевірити їхню коректність. І лише, якщо усі дані користувача задовільняють необхідним умовам, продовжувати обробку даних.

Якщо дані введені некоректно, подальша робота із ними припиняється і форма відправляється користувачу повторно. Поля форми зберігають попередньо введені користувачем дані. Також оновлена форма повинна вивести деталі помилок.

Безпека

На завершення теоретичного вступу в HTML форми розглянемо питання безпеки при роботі із формами.

Необхідний мінімум речей, які нам потрібно застосовувати до наших форм, щоб уникати більшості атак зловмисників, це:

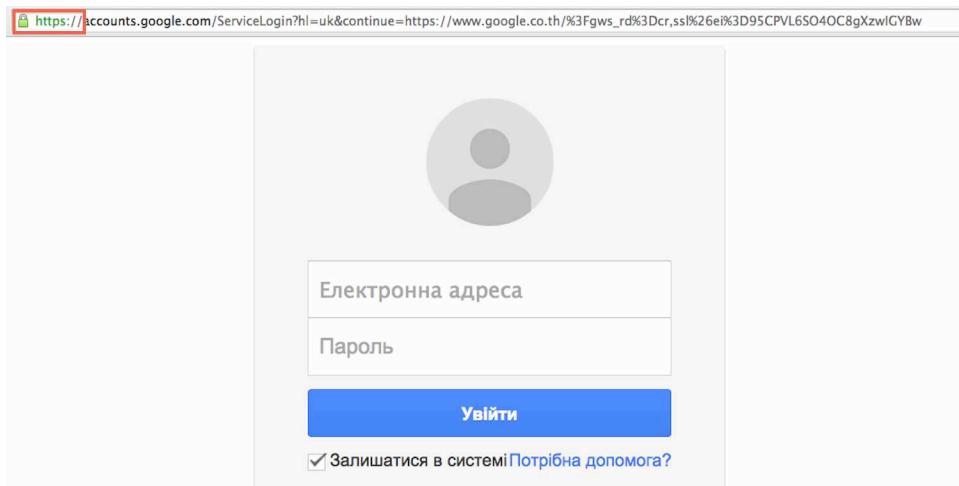
- кодування даних форм при відправці на сервер з допомогою SSL сертифікатів та протоколу [HTTPS²¹²](https://uk.wikipedia.org/wiki/HTTPS);
- захист від підробки запиту від імені користувача; так званий [CSRF²¹³](http://bit.ly/vpcsrc) (Cross Site Request Forgery атака).

²¹²[http://uk.wikipedia.org/wiki/HTTPS](https://uk.wikipedia.org/wiki/HTTPS)

²¹³<http://bit.ly/vpcsrc>

HTTPS

Більшість форм логування та реєстрації (та й усі форми, що передають із веб-сторінки на сервер паролі користувача) в інтернеті працюють не через протокол HTTP, а через HTTPS.



HTTPS протокол для форм із паролем

До сторінок, що обслуговуються через протокол HTTPS застосовують SSL сертифікат, з допомогою якого запит на сервер і відповіді від нього шифруються. Тому, навіть, якщо хтось перехопить дані, вони будуть зашифровані. Крім того HTTPS протокол для нас перевіряє чи ми дійсно спілкуємося із коректним сервером, а не підробкою.

Таким чином, варто дотримуватись простого правила: якщо ви маєте справу із формою, яка відправляє важливі секретні дані, сторінка, на якій дана форма знаходитьться, повинна обслуговуватись через HTTPS протокол.

CSRF

Другим базовим захистом форм є захист проти підробок запитів. Уявіть, що обробник вашої форми запустили не із сторінки форми, а із коду, або з посилання на форумі. А клікнув дане посилання ваш користувач, не знаючи про те, що дане посилання піде на обробник даної форми. В браузері можуть на той час залишатись дані залогованої сесії користувача (наприклад дані cookies) і запит пройде успішно від імені даного залогованого на вашому

сайті користувача. Це може призвести до неочікуваних і невідворотніх змін на вашому веб-сайті.

Більше про CSRF можете почитати [тут²¹⁴](#).

Дана проблема вирішується доволі просто. У себе на форму вставляєте приховане поле із певним автоматично згенерованим кодом, а в обробнику форми перевіряєте, чи отримали правильний код із даної форми. Django уже дає нам можливість генерації такого коду, а також саму валідацію для кожної нашої форми.

Ми повернемось до CSRF проблеми, коли розроблятимо нашу першу форму - форму додавання студента.

...

На цьому закінчуємо із теорією і переходимо до практики:

Форма додавання студента

Почнемо із реалізації форми додавання студента.

Дану форму розробимо найдовшим можливим шляхом: пишучи HTML код форми самостійно з нуля, а також логіку на сервері (включно з валідацією) власними силами.

В наступній секції ми створимо форму редагування студента. І там ми скристаємося Django формами, які для нас зроблять усю важку частину і по генерації HTML коду, і по валідації, і по збереженні змін в базу даних.

Проте, перед тим, як використовувати базу, нам спочатку потрібно розібраться як стріляти із пістолета. Тому, перед використанням усього готовенького, спочатку навчимось самостійно писати форму та її обробник. Це допоможе нам далі зрозуміти, що ж закладено у класи Django форм.

²¹⁴<http://bit.ly/vpcsrc>

Підключаємо шаблон

В попередніх главах ми уже підключили в'юшку і URL шаблон для форми редагування студента. Але тоді ми зробили так, щоб ця в'юшка поверталася статичний текст. Тому давайте переключимо її на використання шаблону.

Для цього спочатку створимо шаблон, скопіювавши його із students_list.html:

Підготовка шаблону students_add.html

```
1 # заходимо в апплікацію, папку із шаблонами
2 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students/templates/students
3
4 # копіюємо students_list.html шаблон
5 $ cp students_list.html students_add.html
```

Тепер підключаємо новий шаблон до в'юшки з додавання студента:

Підключаємо шаблон, модуль views.py

```
1 def students_add(request):
2     return render(request, 'students/students_add.html',
3                  {})
```

Думаю, код вище не потребує пояснень. Подібну річ ми уже робили у функції в'юшки списку студентів.

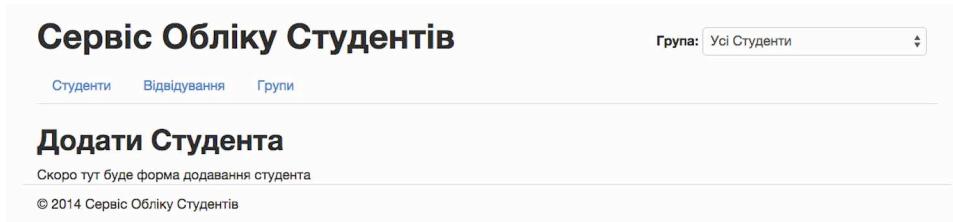
HTML Форма

Настав час зайнятись самим шаблоном і створити форму додавання студента. Відкриваємо шаблон students_add.html в редакторі, оновлюємо заголовки та видаляємо увесь вміст блоку “content”.

Початкові приготування сторінки редагування студента

```
1  {% extends "students/base.html" %}  
2  
3  {% load static from staticfiles %}  
4  
5  {% block meta_title %}Додати Студента{% endblock meta_title %}  
6  
7  {% block title %}Додати Студента{% endblock title %}  
8  
9  {% block content %}  
10  
11  Скоро тут буде форма додавання студента  
12  
13  {% endblock content %}
```

Приблизно ось так виглядатиме наша заготовка в браузері, коли клацнемо на кнопку Додати Студента:



Заготовка сторінки Додати Студента

Дійшли до головного - до самої форми. Додаємо тег “form”, а всередині для кожного поля моделі студента додаємо відповідне поле вводу для користувача:

Базова форма додавання студента

```
1 <form action="{% url "students_add" %}" method="post"
2     enctype="multipart/form-data">
3
4     <div>
5         <label for="first_name">Ім'я</label>
6         <input type="text" value=""
7             name="first_name" id="first_name" />
8     </div>
9     <div>
10        <label for="last_name">Прізвище</label>
11        <input type="text" value=""
12            name="last_name" id="last_name"
13            placeholder="Введіть ваше прізвище" />
14    </div>
15    <div>
16        <label for="middle_name">По-батькові</label>
17        <input type="text" value=""
18            name="middle_name" id="middle_name" />
19    </div>
20    <div>
21        <label for="birthday">Дата Народження</label>
22        <input type="text" value=""
23            name="birthday" id="birthday"
24            placeholder="Напр. 1984-12-30" />
25    </div>
26    <div>
27        <label for="photo">Фото</label>
28        <input type="file" value="" name="photo" id="photo" />
29    </div>
30    <div>
31        <label for="ticket">Білет</label>
32        <input type="text" value=""
33            name="ticket" id="ticket" />
34    </div>
```

```
35 <div>
36     <label for="student_group">Група</label>
37     <select name="student_group" id="student_group">
38         <option value="">Виберіть групу</option>
39         <option value="1">МтМ-1</option>
40         <option value="2">МтМ-2</option>
41     </select>
42 </div>
43 <div>
44     <label for="notes">Додаткові Нотатки</label>
45     <textarea name="notes" id="notes" class="form-control"></textare\46 a>
47 </div>
48 <div>
49     <input type="submit" value="Додати" name="add_button" />
50     <button type="submit" name="cancel_button">Скасувати</button>
51 </div>
52 </form>
```

Пройдемось по усіх фажливих моментах даної форми:

- 1ий рядок: form - тег форми; його атрибут action вказує на цю ж саму сторінку, на якій знаходиться наша форма додавання студента; тобто генератор форми і обробник форми у нас буде той самий - функція в'юшки додавання студента; взалежності від типу запиту (GET чи POST) наша в'юшка робитиме дві різні речі; метод запиту є POST, адже ми змінюватимемо дані на сервері з допомогою даної форми; тип кодування даних перед відправкою на сервер: "multipart/form-data", адже нам потрібно буде відправляти файл зображення студента;
- 4ий: div - кожне поле складається, як мінімум, із двох тегів: мітки поля (label) та самого поля вводу; кожну таку пару ми огортаємо в блоковий елемент div, щоб візуально відокремити поля і поставити їх в окремі рядки;
- 5ий: label - для кожного поля вводу додаємо цей тег; він додає пояснення про призначення поля, а також, з допомогою атрибуту "for", прив'язує

фокус поля до кліка по мітці; “for” атрибут містить “id” поля, якому відповідає тег “label”;

- 6ий: `input` - поле текстової стрічки для поля Ім’я студента; значення поля (`value`) порожнє;
- 13ий: `placeholder` - цей атрибут вставляє в поле текст, який ми зазвичай використовуємо для пояснення користувачу, що саме потрібно ввести у дане поле; як тільки користувач починає вводити текст у дане поле, `placeholder` стрічка одразу зникає; на домашнє завдання: додайте атрибут “placeholder” до усіх полів форми;
- 28ий: `input[type=file]` - поле типу “file” призначено для відправки файлів на сервер;
- 37ий: `select` - даний тег використовуємо для відображення списку існуючих груп в аплікації;
- 38ий: `option` - з допомогою даного тегу набиваємо список груп; в даний момент він у нас статичний, пізніше ми його динамізуємо існуючими групами з бази;
- 45ий: `textarea` - тег для вводу багатострічкового тексту; замість атрибути “`value`”, значення даного поля вводиться всередині тегу;
- 49ий: `input` - поле типу “submit” призначено для відправки даних форми на сервер; по атрибуту “`name`” можемо на сервері ідентифікувати, яка саме кнопка була натиснута при відправці даних; атрибут “`value`” використовується для тексту на кнопці; також відправити дані на сервер можна натиснувши кнопку “Enter” маючи сфокусованим одне із полів форми; при цьому буде автоматично запущена перша кнопка на формі;
- 50ий: `button` - аналогічно до поля `input` з типом “`submit`”, можна використовувати тег `button`; він аналогічний по функціоналу, але кращий у випадку, якщо потрібно додати кастомні стилі до кнопки.

Зберігаємо оновлений шаблон і оновляємо сторінку в браузері:

Додати Студента

Ім'я

Прізвище Введіть ваше прізви

По-батькові

Дата Народження Напр. 1984-12-30

Фото Choose File No file chosen

Білет

Група

Додаткові Нотатки

Базова форма додавання студента

Уже досить непогано! Тому переходимо до в'юшки і попрацюємо із Python кодом.

Оновлюємо в'юшку

Першим ділом оновимо поле “Група” списком реальних груп із бази даних. Для цього передамо з в'юшки в шаблон список усіх груп:

Передаємо групи в шаблон, `students.py` модуль в пакеті `views`.

```

1 from ..models import Student, Group
2
3 def students_add(request):
4     return render(request, 'students/students_add.html',
5                  {'groups': Group.objects.all().order_by('title')})

```

Буквально одним рядочком ми отримали список усіх груп в базі даних посортуваних по імені і передали їх під ключем ‘groups’. Таким чином можемо тепер ним скористатись в шаблоні:

Список існуючих груп готовий

```
1 <select name="student_group" id="student_group">
2   <option value="">Виберіть групу</option>
3   {% for group in groups %}
4     <option value="{{ group.id }}">{{ group.title }}</option>
5   {% endfor %}
6 </select>
```

В 3му рядку ми пробігаємось в циклі по списку “groups”. В тілі даного циклу ми створюємо теги “option”, значеннями яких є унікальні ідентифікатори груп (id), а ім’я групи використовуємо для візуального представлення користувачеві.

Спробуйте тепер перевантажити вашу сторінку і переконайтесь, що випадайка з групами відображає існуючі групи з бази даних.

В даний момент наша в’юшка вміє лише показувати форму додавання студента, але ще не вміє обробляти пост даної форми. Наша задача полягає в тому, щоб навчити дану в’юшку:

- перевіряти чи відбувся пост форми;
- якщо відбувся, тоді перевірити вхідні дані;
- якщо дані правильні, створити і зберегти нового студента в базу;
- вернутись на список із студентами;
- якщо дані неправильні, вернути форму і вказати користувачеві на вказані помилки;
- при цьому бажано залишити попередні значення полів введені користувачем;
- якщо пост форми відбувся, але кнопка “Відмінити” була натиснута, тоді просто переводимо користувача на сторінку із списком студентів;
- якщо ж поста форми не було, просто показуємо початкову порожню форму.

Давайте почнемо малими кроками оновлювати нашу в’юшку. Найкращим варіантом буде почати із метакоду - додати коментарі щодо загальної логіки (так як ми це описали у списку вище) до коду в’юшки:

Реалізуємо в'юшку на метакоді

```
1 def students_add(request):
2     # Якщо форма була запошена:
3
4         # Якщо кнопка Скасувати була натиснута:
5
6             # Повертаємо користувача до списку студентів
7
8         # Якщо кнопка Додати була натиснута:
9
10        # Перевіряємо дані на коректність та збираємо помилки
11
12        # Якщо дані були введені некоректно:
13            # Віддаємо шаблон форми разом із знайденими помилками
14
15        # Якщо дані були введені коректно:
16            # Створюємо та зберігаємо студента в базу
17
18            # Повертаємо користувача до списку студентів
19
20    # Якщо форма не була запошена:
21        # повертаємо код початкового стану форми
22    return render(request, 'students/students_add.html',
23                  {'groups': Group.objects.all().order_by('title')})
```

Для чого ми це робимо? Для того, щоб отримати повну картину необхідної логіки, і вже потім, маючи загальний потік дій у в'юшці, по-трохи заповнювати реальним кодом. Таким чином, ми ітеративно розширюватимемо нашу функцію щораз складнішим кодом.

Рекомендую використовувати даний підхід “метакоду” у власній щоденній

практиці. Він допоможе вам із самого початку краще моделювати ваш код, API та архітектуру ваших проектів.

Почнемо із найпростіших речей: перевірка на постформи, а також збереження студента. А валідацію даних залишимо на кінець, оскільки це буде найскладніша частина логіки нашої в'юшки.

Ось як виглядатиме код функції з оновленими умовами:

Додаємо перевірку на сабміт форми та додаємо студента

```
25             ticket=request.POST['ticket'],
26             student_group=
27                 Group.objects.get(pk=request.POST['student_gro\
28 up']),
29             photo=request.FILES['photo'],
30         )
31
32         # save it to database
33         student.save()
34
35         # redirect user to students list
36         return HttpResponseRedirect(reverse('home'))
37
38     else:
39         # render form with errors and previous user input
40         return render(request, 'students/students_add.html',
41                     {'groups': Group.objects.all().order_by('title'),
42                      'errors': errors})
43     elif request.POST.get('cancel_button') is not None:
44         # redirect to home page on cancel button
45         return HttpResponseRedirect(reverse('home'))
46     else:
47         # initial form render
48         return render(request, 'students/students_add.html',
49                     {'groups': Group.objects.all().order_by('title')})
```

Ми замінили практично весь метакод окрім валідації даних. То ж, що саме ми зробили:

- 2ий рядок: імпортуємо HttpResponseRedirect клас, який використовува-
тимемо для редіректу браузера на сторінку із списком студентів при
успішній обробці форми;
- 11ий: POST - перевіряємо чи форма була відправлена; в такому випадку
метод запиту буде “POST”;

- 13ий: перевіряємо чи була натиснута кнопка Додати Студента; так, кнопки форми також потрапляють в словник POST об'єкта запиту; ось чому важливо коректно встановити атрибут “name” для тегу кнопки;
- 15ий: залишаємо коментар TODO (завдання на майбутнє), щоб пізніше не забути додати сюди логіки по валідації даних з форми; ми визначили змінну-словник errors, в яку пізніше додаватимемо знайдені помилки; а поки він у нас порожній;
- 18ий: перевіряємо чи немає помилок валідації даних;
- 20ий: якщо помилок таки немає (тобто словник errors порожній), створюємо об'єкт студента; йому передаємо дані із словника POST; оскільки наша форма має метод 'POST', то і дані на сервер прийшли у словнику POST; якщо б форма працювала через метод 'GET', тоді, відповідно, ми б мали працювати на сервері із атрибутом GET об'єкта запиту;
- 26ий: як ви вже знаєте, поле моделі ForeignKey вимагає в якості значення об'єкт класу моделі; саме тому в даному рядку ми отримуємо з бази даних групу ідентифіковану через поле 'student_group'; дане поле є рівним "ID" полю групи; метод "get" поверне нам єдиний унікальний об'єкт форми за даним "ID";
- 29ий: файли у формі приходять до нас по-іншому, ніж решта полів; а саме в атрибуті запиту "FILES";
- 33ій: зберігаємо новоствореного студента в базу даних; на цьому етапі, якщо передані дані є некоректними, або певних обов'язкових полів моделі бракує, отримаємо помилку; тому для тесту даного коду, поки у нас немає валідації, введіть усі поля на формі правильним чином;
- 36ий: після успішного збереження студента перенаправляємо користувача на список студентів; для цього повертаємо із нашої функції об'єкт класу HttpResponseRedirect з переданим йому параметром: адресою сторінки, на яку перенаправити користувача; дану адресу ми сформували з допомогою функції reverse, яку імпортували на початку модуля; мінімальний набір аргументів даної функції - це назва URL шаблона, який ми задекларували у модулі urls.py; функція reverse є аналогом шаблонного тегу url;
- 38ий: починаємо гілку коду, яка виконується при невдалій валідації вводу від користувача;

- 40ий: у цьому випадку ми повторно показуємо форму додавання студента, але цього разу ще й передаємо в шаблон наш словник із помилками “errors”; пізніше, коли матимемо валідацію, також оновимо шаблон, щоб він правильно використовував дані помилки для відображення користувачеві;
- 43ій: перевіряємо чи користувач натиснув кнопку “Скасувати”;
- 45ий: якщо так, тоді просто перенаправляємо користувача на сторінку із списком студентів;
- 46ий: наша початкова гілка: відображаємо початковий стан форми, якщо тип запиту не є “POST”.

HTTP редірект - це перенаправлення браузера на іншу сторінку. Всередині даний редірект відбувається з допомогою HTTP заголовка “Location”, якому передають, в якості значення, URL адресу на іншу сторінку. Даний заголовок встановлюється сервером і зазвичай використовується після успішної обробки форми.

Спробуйте тепер перезавантажити вашу сторінку з формою додавання студента, заповнити усі поля коректними даними (поле дати заповніть у форматі [рік]-[місяць]-[день], напр. “1992-10-30”) та натисніть кнопку Додати. Ви повинні перейти на сторінку списку студентів і там, в самому кінці списку, знайти вашого новододаного студента. Зайдіть в адмін частину Django сайту і переконайтесь, що усі введені вами поля є присутні в об’єкті студента.

Натомість отримали помилку “CSRF verification failed. Request aborted.”? Так і було задумано ;-) Як ми уже з вами розібралися в першій секції даної глави, Django фреймворк обов’язково перевіряє кожну форму на наявність певного коду, щоб уникати CSRF хаку. Щоб доробити нашу форму для даної валідації просто додайте ось цей тег де-небудь всередині тегу form: “`{% csrf_token %}`”. Він вставить для нас приховане поле із наперед згенерованим кодом для подальшої верифікації при пості форми. Тепер має усе працювати.

Також спробуйте скористатись кнопкою Скасувати. Ви маєте повернутись на список студентів, але цього разу без жодних змін у базу даних.

Якщо щось не виходить як описано вище, перегляньте код наведений в книзі, перегляньте ваш власний код і спробуйте запустити. В кінці кінців, гляньте в код, що йде разом із книгою і порівняйте, що у вас не так.

Валідація даних

Переходимо до валідації. Валідація є однією із найвідповідальніших частин роботи із формою, і, водночас, однією із найважчих. Якщо недоглядіти і пропустити дрібний момент, то можна залишитись із поламаною базою чи веб-сайтом.

Для більш-менш пристойної валідації на нашій формі нам потрібно:

- позначити обов'язкові поля на формі;
- перевірити поля на коректно введені дані;
- кожне невірно введене поле, або поле, яке є обов'язковим для введення, але користувач його пропустив, повинне бути згаданим в словнику 'errors'; в даному словнику ми вписуватимемо ключі одноіменні з назвами полів, а значеннями будуть стрічки тексту (повідомлення) для користувача з деталями помилок;
- якщо є хоча б одна помилка, тоді віддаємо шаблон форми;
- при цьому показуємо помилки біля кожного поля із некоректними даними;
- а також, для зручності користувача, зберігатимемо в полях форми попередньо введені дані; інакше користувачеві прийдеться щоразу вводити навіть ті дані, які були введені коректно попереднього разу.

Почнемо із простого, позначимо усі обов'язкові поля на формі додатковою зірочкою, що йде одразу після тексту мітки:

Відмічаємо зірочкою поле Ім'я

```
1 <div>
2     <label for="first_name">Ім'я*</label>
3     <input type="text" value=""
4         name="first_name" id="first_name" />
5 </div>
```

В другому рядку можете бачити символ зірочки одразу після слова “Ім'я”. Додайте також таку зірочку до усіх обов'язкових полів моделі Student: Ім'я, Прізвище, Дата Народження, Білет та Група.

Тепер користувач знатиме, що дані поля є обов'язковими і це збереже йому кілька секунд повторної відправки даних форми.

Наступним кроком додамо в нашу в'юшку Python код, який перевірятиме усі обов'язкові поля і повернеть помилки, якщо їх бракуватиме:

Валідуємо обов'язкові поля

```
15     # errors collection
16     errors = {}
17     # validate student data will go here
18     data = {'middle_name': request.POST.get('middle_name'),
19             'notes': request.POST.get('notes')}
20
21     # validate user input
22     first_name = request.POST.get('first_name', '').strip()
23     if not first_name:
24         errors['first_name'] = u"Ім'я є обов'язковим"
25     else:
26         data['first_name'] = first_name
27
28     last_name = request.POST.get('last_name', '').strip()
29     if not last_name:
30         errors['last_name'] = u"Прізвище є обов'язковим"
31     else:
```

```
32         data['last_name'] = last_name
33
34     birthday = request.POST.get('birthday', '').strip()
35     if not birthday:
36         errors['birthday'] = u"Дата народження є обов'язковою"
37     else:
38         data['birthday'] = birthday
39
40
41     ticket = request.POST.get('ticket', '').strip()
42     if not ticket:
43         errors['ticket'] = u"Номер білета є обов'язковим"
44     else:
45         data['ticket'] = ticket
46
47     student_group = request.POST.get('student_group', '').st\
48     rip()
49     if not student_group:
50         errors['student_group'] = u"Оберіть групу для студен\
51     та"
52     else:
53         data['student_group'] = Group.objects.get(pk=student\
54     _group)
55
56     photo = request.FILES.get('photo')
57     if photo:
58         data['photo'] = photo
59
60     # save student
61     if not errors:
62         student = Student(**data)
63         student.save()
64
65     # redirect to students list
66     return HttpResponseRedirect(reverse('home'))
```

```
67     else:
68         # render form with errors and previous user input
69         return render(request, 'students/students_add.html',
70                     {'groups': Group.objects.all().order_by('title'),
71                      'errors': errors})
```

Вище наведено лише частину в'юшки додавання студента. Ту частина, де ми мали “TODO” нотатку для подальшого коду валідації.

Давайте пройдемось детальніше по останніх оновленнях:

- 16ий рядок: декларуємо змінну-словник для збору знайдених помилок;
- 18ий: в словник data будемо збирати коректні і нормалізовані дані з форми; ми одразу у ньому прописали два поля, які не потребують додаткової валідації: Нотатки та По-батькові;
- 22ий: витягуємо ім'я студента із форми (атрибути POST) і обрізаємо по краях непотрібні пробіли; таким чином не пропускаємо від користувача стрічки, що міститиме лише пробіли;
- 23ій: перевіряємо чи стрічка не є порожньою;
- 24ій: якщо порожня, тоді додаємо в словник errors під ключем назви поточного поля стрічку-повідомлення про помилку;
- 26ий: якщо поле імені введене коректно, то додаємо його до словника data, на базі якого пізніше створимо об'єкт студента;
- 28ий: подібну річ проробляємо із полями Прізвище, День Народження, Білет, Група та полем Фото;
- 62ий: як бачите, тепер ми не витягуємо усі змінні ще раз із POST, а натомість використовуємо уже готовий для нас словник data; з допомогою спеціального синтаксису двох зірочок ми можемо функції передати розгорнутий в keyword аргументи словник; ефект буде такий самий як і при передачі напряму аргументів “first_name=” і т.д.;
- 71ий: якщо є помилки, то передаємо їх в шаблон із формами.

Тепер давайте оновимо наш шаблон так, щоб скористатись переданими помилками:

Виводимо помилки на форму

```

1 <div>
2   <label for="first_name">Ім'я*</label>
3   <input type="text" value=""
4     name="first_name" id="first_name" />
5   <span class="help-block">{{ errors.first_name }}</span>
6 </div>

```

До поля “Ім’я” ми додали ще один тег. Цього разу це span із класом `help-block`²¹⁵. Всередині нього можете бачити змінну “`errors.first_name`”. Із словника `errors` ми пробуємо витягнути ключ “`first_name`”. Якщо його не існує, шаблон тихо проігнорує дану проблему без будь-яких помилок про неіснуючий ключ.

Додайте самостійно подібний span тег до усіх полів, при цьому змінивши ключ на назву відповідного поля.

Після цього оновіть сторінку із формою в браузері. Не вводіть ніяких даних та натисніть кнопку Додати. В результаті даної дії ви повинні залишитись на формі, але тепер форма дає вам знати деталі помилок вводу:

Базова валідація форми працює

²¹⁵<http://getbootstrap.com/css/#forms-control-validation>

Гаразд. Базова валідація є, показується на формі для користувача і форму ми не пропускаємо на запис даних, якщо є некоректні дані. Але нам бракує ще більш розширених перевірок. Зокрема, для поля дати нам потрібно перевіряти чи формат стрічки є правильним.

Для поля групи потрібно перевіряти чи група існує в базі. Інакше наша в'юшка просто ламатиметься замість того, щоб повернати форму із гарними повідомленнями про помилки.

Ось повний оновлений код функції в'юшки форми додавання студента:

Код із розширеною валідацією даних

```
1 # новий імпорт в модулі
2 from datetime import datetime
3
4 def students_add(request):
5     # was form posted?
6     if request.method == "POST":
7         # was form add button clicked?
8         if request.POST.get('add_button') is not None:
9             # errors collection
10            errors = {}
11
12            # data for student object
13            data = {'middle_name': request.POST.get('middle_name'),
14                    'notes': request.POST.get('notes')}
15
16            # validate user input
17            first_name = request.POST.get('first_name', '').strip()
18            if not first_name:
19                errors['first_name'] = u"Ім'я є обов'язковим"
20            else:
21                data['first_name'] = first_name
22
23            last_name = request.POST.get('last_name', '').strip()
24            if not last_name:
25                errors['last_name'] = u"Прізвище є обов'язковим"
```

```
26         else:
27             data[ 'last_name' ] = last_name
28
29             birthday = request.POST.get( 'birthday', '' ).strip()
30             if not birthday:
31                 errors[ 'birthday' ] = u"Дата народження є обов'язково\
32                 ю"
33             else:
34                 try:
35                     datetime.strptime(birthday, '%Y-%m-%d')
36                 except Exception:
37                     errors[ 'birthday' ] = \
38                         u"Введіть коректний формат дати (напр. 1984-\\
39                         12-30)"
40             else:
41                 data[ 'birthday' ] = birthday
42
43             ticket = request.POST.get( 'ticket', '' ).strip()
44             if not ticket:
45                 errors[ 'ticket' ] = u"Номер білета є обов'язковим"
46             else:
47                 data[ 'ticket' ] = ticket
48
49             student_group = request.POST.get( 'student_group', '' ).st\
50             rip()
51             if not student_group:
52                 errors[ 'student_group' ] = u"Оберіть групу для студен\
53                 та"
54             else:
55                 groups = Group.objects.filter(pk=student_group)
56                 if len(groups) != 1:
57                     errors[ 'student_group' ] = u"Оберіть коректну гру\
58                     пу"
59             else:
60                 data[ 'student_group' ] = groups[0]
```

```
61
62
63     photo = request.FILES.get('photo')
64     if photo:
65         data['photo'] = photo
66
67     # save student
68     if not errors:
69         student = Student(**data)
70         student.save()
71
72     # redirect to students list
73     return HttpResponseRedirect(reverse('home'))
74 else:
75     # render form with errors and previous user input
76     return render(request, 'students/students_add.html',
77                   {'groups': Group.objects.all().order_by('title'),
78                    'errors': errors})
79 elif request.POST.get('cancel_button') is not None:
80     # redirect to home page on cancel button
81     return HttpResponseRedirect(reverse('home'))
82 else:
83     # initial form render
84     return render(request, 'students/students_add.html',
85                   {'groups': Group.objects.all().order_by('title')})
```

То ж, що змінилось в нашій функції:

- 34ий рядок: ми оновили гілку валідації поля Дати Народження; тепер ми не лише перевіряємо на наявність дане поле, але і на правильний формат; для цього ми використовуємо напередодні імпортований модуль `datetime` (це вбудований модуль мови Python) та користуємось функцією `strptime`, щоб спробувати перетворити стрічку в Python `datetime` об'єкт; якщо стрічка не задовільняє вказаного формату, дана функція викине

помилку; саме тому ми огорнули її виклик в “try/except” гілку; більше про роботу із датами в Python можете почитати [тут²¹⁶](#);

- 37ий: якщо передана стрічка дати неправильного формату, ми видаємо окреме повідомлення і нагадуємо про правильний формат для поля дати;
- 55ий: ми трохи оновили роботу із полем групи; ми дістаємо групу з бази даних через метод filter, що дозволить нам уникнути помилок в Python коді, якщо за даною ID (student_group) не виявиться групи в базі даних; груп із даною ID повинно бути рівно 1; якщо це не так, видаємо користувачу повідомлення, що група вибрана невірно.

Решта коду в даній функції залишилась незмінною.

На домашнє завдання реалізувати валідацію поля Фото. Файл фото повинен бути не більше 2 мегабайт у розмірі та бути дійсно файлом зображення. Підказка: найпримітивніший варіант валідації типу файла може бути по розширенню назви файла. Складніший і більш надійний метод буде валідація через вміст файла. Це можна зробити або з допомогою Python бібліотеки PIL, або використовуючи Django валідатор. Почніть із простішого варіанту, а далі спробуйте складніший. Складніший варіант вимагатиме від вас немалого часу для реалізації.

...

Вам зручно кожного разу перенабирати усі поля форми після кожної невдалої валідації даних? Мені також ні. Тому давайте, зробимо так, щоб усі попередньо введені дані залишались у формі, якщо валідація даних не пройшла.

Це робиться доволі просто. Генерація шаблону відбувається при обробці того ж запиту, що і запуск в'юшки. Відповідно, об'єкт запиту (request) буде однаковим як для в'юшки, так і для шаблона. Отже, маємо в шаблоні доступ до усіх даних з форми. Давайте вставимо в значення полів форми ті дані, що є в запиті:

²¹⁶<http://asvetlov.blogspot.com/2012/10/format.html>

Зберігаємо дані користувача на формі після невдалої валідації

```

1 <div>
2   <label for="first_name">Ім'я*</label>
3   <input type="text" value="{{ request.POST.first_name }}"
4     name="first_name" id="first_name" />
5   <span class="help-block">{{ errors.first_name }}</span>
6 </div>

```

Як бачите все доволі просто. Змінну “{{ request.POST.first_name }}” ми вставили значенням в атрибут “value” тегу input. Проробіть подібне оновлення усіх інших полів самостійно.

Думаю вас мало озадачити поле Група?...

Тег select заслуговує окремої уваги. Логіка із вибору потрібної опції в даному тезі потребує трохи більше коду:

Встановлюємо попереднє значення у полі Група

```

1 <select name="student_group" id="student_group"
2   class="form-control">
3   <option value="">Виберіть групу</option>
4   {% for group in groups %}
5     <option value="{{ group.id }}" {% if group.id == request.POST.student_group|add:0 %}selected="1"{% endif %}>{{ group.title }}</option>
6   {% endfor %}
7 </select>

```

В тегу option ми додали тег умови. Якщо поточна група ітерації є рівна попередньо вибраній групі, тоді ми маркуємо поточний тег option як “selected”. Зверніть увагу, що ми порівнюємо “id” об’єкта групи (дане поле є числом) із ключем “student_group” (який є юнікодовою стрічкою), що знаходитьться в пості запиту. Тому, щоб перевести обидва операнди операції порівняння, ми додаємо фільтр “add:0” до змінної “student_group”. Даний фільтр спробує додати до стрічки “student_group” нуль і результат поверне у вигляді числа. Цей хак ми застосували спеціально, щоб отримати з обох сторін знаку порівняння операнди типу integer.

Полю `input` типу “`file`” не можна ніяким чином встановити значення, окрім як користувач сам його вибере. Це обмеження зроблено спеціально для безпеки в інтернеті. Інакше ми б могли із сервера чи Javascript коду самостійно приєднати файл на формі із машини нашого користувача. Доступ до вашої файлової системи будь-яким веб-сайтом в інтернеті, самі розумієте, був би поганою ідеєю. Для обходу даного обмеження можна реалізувати Ajax форми без перевантаження сторінки, що дозволить користувачу не приєднувати файл кожного разу при введені некоректних даних на формі.

Таким чином ми з вами повністю розробили HTML код форми та логіку з її обробки. Дані логіка правильно обробляє обидві кнопки на формі, перевіряє дані форми, виводить помилки про некоректні дані, створює та зберігає студента та виводить попередні дані користувача у випадку помилок.

Наша форма практично готова. Залишилось зробити дві речі:

- додати глобальні статусні повідомлення про результати дій користувача із формою;
- заставити форму виглядати трохи краще із стилями від Twitter Bootstrap.

Статусні повідомлення

На даний момент, коли ви додаєте студента або скасовуєте зміни, ви переходите на сторінку із списком студентів. Проте в обидвох випадках не зовсім зрозуміло, що насправді відбулось. Було б добре на цій сторінці показати повідомлення користувачеві про результат виконаних дій.

Саме це ми і зробимо у цій секції. Реалізуємо статусні повідомлення, які, на даний момент, працюватимуть згідно наступних тьох сценаріїв:

- якщо форма була відправлена на сервер, але виникли помилки валідації даних, ми показуємо статусне повідомлення і вказуємо користувачеві, що знайдено помилки на формі;

- якщо студент успішно доданий, показуємо повідомлення на сторінці із списком студентів;
- якщо натиснута кнопка “Скасувати”, показуємо повідомлення, що зміни скасовано.

Реалізуємо дані повідомлення своїми силами. Для цього відкоментуємо наш давно забутий віджет “alert”, який ми застилили згідно із Twitter Bootstrap алертами²¹⁷.

Ось знайдений і відкоментований “alert” елемент в шаблоні base.html:

Статусне повідомлення в base.html шаблоні

```
1      <!-- Start Content Columns -->
2      <div class="row" id="content-columns">
3          <div class="col-xs-12" id="content-column">
4
5              <div class="alert alert-warning" role="alert">Статусне повід\
6 омлення</div>
7
8              <h2>{% block title %}{% endblock title %}</h2>
9              {% block content %}{% endblock content %}
10
11         </div>
12     </div>
13     <!-- End Content Columns -->
```

В 5му рядку, як бачите, у нас є “Статусне повідомлення”. Але поки воно лише статичне і постійно показуватиметься на сторінці.

Найпростішим підходом, щоб передавати повідомлення між сторінками при редіректах, буде використання URL параметрів. Більшість запитів будуть запитами типу GET, відповідно, можемо перевіряти чи параметр присутній в запиті. Якщо так - показуємо наше повідомлення, а значення параметра вставляємо всередину тега алерта:

²¹⁷<http://getbootstrap.com/components/#alerts>

Статусне повідомлення в base.html шаблоні

```
1  <!-- Start Content Columns -->
2  <div class="row" id="content-columns">
3      <div class="col-xs-12" id="content-column">
4
5          {% block status_message %}
6          {% if request.GET.status_message %}
7              <div class="alert alert-warning" role="alert">{{ request.GET\
8 .status_message }}</div>
9          {% endif %}
10         {% endblock %}
11
12         <h2>{% block title %}{% endblock title %}</h2>
13         {% block content %}{% endblock content %}
14
15     </div>
16 </div>
17 <!-- End Content Columns -->
```

Як бачите, змінну, в якій передаватимемо статусне повідомлення ми назвали “status_message”. Якщо дана змінна присутня в запиті (“request.GET”), тоді показуємо елемент із алертом, а значення цієї змінної вставляємо всередині тегу. Окрім того, зауважте, що весь алерт є тепер огорнутим в іменований блок “status_message”. Це нам пригодиться пізніше, коли треба буде по іншому вставляти дане повідомлення на тих сторінках, де у нас немає редіректу. А саме, на сторінці з формою додавання студента, коли валідація даних не проходить.

Тепер, коли наші шаблони готові до відображення повідомень, залишилось навчити наші в'юшки редіректити на сторінки і при цьому передавати додатковий параметр “status_message”.

Почнемо із редіректів, що відбуваються при успішному додаванні студента та при натисканні кнопки Скасувати:

Додаємо status_message до в'юшки students_add

```
1 def students_add(request):
2     # was form posted?
3     if request.method == "POST":
4         # was form add button clicked?
5         if request.POST.get('add_button') is not None:
6             # errors collection
7             errors = {}
8
9             # data for student object
10            data = {'middle_name': request.POST.get('middle_name'),
11                    'notes': request.POST.get('notes')}
12
13            # validate user input
14            first_name = request.POST.get('first_name', '').strip()
15            if not first_name:
16                errors['first_name'] = u"Ім'я є обов'язковим"
17            else:
18                data['first_name'] = first_name
19
20            last_name = request.POST.get('last_name', '').strip()
21            if not last_name:
22                errors['last_name'] = u"Прізвище є обов'язковим"
23            else:
24                data['last_name'] = last_name
25
26            birthday = request.POST.get('birthday', '').strip()
27            if not birthday:
28                errors['birthday'] = u"Дата народження є обов'язково\
29                ю"
30            else:
31                try:
32                    datetime.strptime(birthday, '%Y-%m-%d')
33                except Exception:
34                    errors['birthday'] = \
```

```
35                                     u"Введіть коректний формат дати (напр. 1984-\n36 12-30)"\n37\n38     else:\n39         data['birthday'] = birthday\n40\n41     ticket = request.POST.get('ticket', '').strip()\n42     if not ticket:\n43         errors['ticket'] = u"Номер білета є обов'язковим"\n44     else:\n45         data['ticket'] = ticket\n46\n47     student_group = request.POST.get('student_group', '').st\\\n48     rip()\n49     if not student_group:\n50         errors['student_group'] = u"Оберіть групу для студен\\\n51     та"\n52     else:\n53         groups = Group.objects.filter(pk=student_group)\n54         if len(groups) != 1:\n55             errors['student_group'] = u"Оберіть коректну гру\\\n56     пу"\n57     else:\n58         data['student_group'] = groups[0]\n59\n60\n61     photo = request.FILES.get('photo')\n62     if photo:\n63         data['photo'] = photo\n64\n65     # save student\n66     if not errors:\n67         student = Student(**data)\n68         student.save()\n69\n70     # redirect to students list
```

```
70         return HttpResponseRedirect(
71             u'?status_message=Студента успішно додано!' %
72             reverse('home'))
73     else:
74         # render form with errors and previous user input
75         return render(request, 'students/students_add.html',
76                       {'groups': Group.objects.all().order_by('title'),
77                        'errors': errors})
78     elif request.POST.get('cancel_button') is not None:
79         # redirect to home page on cancel button
80         return HttpResponseRedirect(
81             u'?status_message=Додавання студента скасовано!' %
82             reverse('home'))
83     else:
84         # initial form render
85         return render(request, 'students/students_add.html',
86                       {'groups': Group.objects.all().order_by('title')})
```

Я навів повний код функції в'юшки додавання студента. Але змінені були лише два рядочки:

- 70ий рядок: до HttpResponseRedirect стрічки додали '?status_message=Студента успішно додано!'; зробили ми це з допомогою інтерполяції стрічок в мові Python;
- 80ий: таким самим чином ми додали повідомлення при натиску кнопки Скасувати.

На домашнє завдання: додайте до статусного повідомлення про успішне створення студента ще й повне ім'я новоствореного об'єкта.

Ось як виглядатиме повідомлення про скасоване додавання студента:

Сервіс Обліку Студентів

Група: Усі Студенти

Студенти Відвідування Групи

Добавлення студента скасовано!

База Студентів

Додати Студента

#	Фото	Прізвище	Ім'я	№ Білету	Дії
1		Podoba	Vitaliy	1234	<button>Дія ▾</button>

Статусне повідомлення

Залишився ще один сценарій: статусне повідомлення про знайдені помилки на формі. Тут маємо дещо складнішу ситуацію, адже при неправильних даних ми не робимо редіректу, відповідно, і не можемо встановити URL параметр.

Зробіть паузу і подумайте як би ви реалізували дане повідомлення...

А тепер дивіться, який варіант я пропоную використати. Ми уже огорнули наше статусне повідомлення в тег блоку, тому можемо його повністю перекривати в кінцевому шаблоні. Це ми і зробимо в шаблоні `students_add.html`. В даному шаблоні ми також маємо доступну змінну `"errors"`, згідно якої можемо визначати чи показувати повідомлення чи ні. А саме повідомлення можемо вписати статичним текстом, адже воно завжди матиме єдину причину: помилки на формі. Додайте наступний блок де-небудь в шаблоні `students_add.html` на кореневому рівні:

Додаємо статусне повідомлення для форми додавання студента

```

1  {% block status_message %}
2    {% if errors %}
3      <div class="alert alert-warning" role="alert">
4        Будь-ласка, виправте наступні помилки
5      </div>
6    {% endif %}
7  {% endblock status_message %}
```

Думаю додаткових пояснень тут не потрібно. Таким чином маємо реалізовані усі три сценарія із статусними повідомленнями.

На домашнє завдання пропоную вам переробити статусні повідомлення з використанням вбудованої Django аплікації “django.contrib.messages”. Вони спеціально призначені для даного завдання та надають як серверну логіку, так і тег для шаблонів, щоб з легкістю і гнучко відображати статусні повідомлення. Найпоширеніше їхне застосування - це робота із формами.

Стилі Twitter Bootstrap

На завершення даної секції приведемо нашу форму додавання студента до гарного вигляду. Як завжди, скористаємося готовенькими стилями з чудового фреймворка [Twitter Bootstrap²¹⁸](#).

У нас є кілька вимог до вигляду форми:

- збільшити вертикальні відстані між полями форми;
- підрівняти мітки і поля між собою;
- зробити ширину полів однаковою;
- форма має змінювати ширину відповідно до ширини вікна браузера; тобто гарно виглядати і на десктопі, і на мобільному;
- мітки мають бути в одному рядку із полями;
- повідомлення про помилки мають бути червоного кольору, сигналізуючи, таким чином, про несправність;
- кнопку Додати зробити темно синьою, а кнопку Скасувати у вигляді лінка.

Для початку по-максимуму скористаємося маркапом Twitter Bootstrap форм і додамо усі необхідні класи і огортаючі елементи нашій формі:

- “form-horizontal” клас до тегу form;
- “form-control” клас до усіх полів форми;

²¹⁸<http://getbootstrap.com/css/#forms>

- “form-group” до огортаючих div елементів наших полів;
- також до “form-group” класу додамо “has-error” клас у випадку, якщо поле має помилку під час валідації;
- до міток (тег “label”) додамо клас “control-label”;
- також до міток додамо позиційний клас “col-sm-2”, вказавши займати їм 2 частини із 12-ти;
- а 10 частин займатимуть поля; для цього огорнемо наші теги полів (теги input, select, textarea) та тег з помилкою “span.help-block” в додатковий тер div, якому присвоємо клас “col-sm-10”;
- класи “btn” і “btn-primary” до кнопки “Додати”; це надасть їй гарного синього кольору і дефолтних стилів бібліотеки;
- класи “btn” і “btn-link” до кнопки “Скасувати”; після цього вона матиме вигляд лінка; це непогано підходить для кнопок з відмінами дій.

Нічого свого ми не придумували, а лише узяли класи із Twitter Bootstrap стилів для форм і додали до наших тегів. Ось кінцевий результат оновленої форми:

Наша форма із Twitter Bootstrap стилями

```
1 <form action="." method="post" enctype="multipart/form-data"
2     role="form" class="form-horizontal">
3     {% csrf_token %}
4
5     <div class="form-group {% if errors.first_name %}has-error{% endif\%
6     %}">
7         <label for="first_name" class="col-sm-2 control-label">Ім'я*</la\bel>
8             <div class="col-sm-10">
9                 <input type="text" value="{{ request.POST.first_name }}"
10                    name="first_name" id="first_name"
11                    class="form-control" />
12                 <span class="help-block">{{ errors.first_name }}</span>
13             </div>
14         </div>
15     </div>
16     <div class="form-group {% if errors.last_name %}has-error{% endif \
```

```
17 %}">
18     <label for="last_name" class="col-sm-2 control-label">Прізвище*< \
19 /label>
20     <div class="col-sm-10">
21         <input type="text" value="{{ request.POST.first_name }}" \
22             name="last_name" id="last_name" \
23             placeholder="Введіть ваше прізвище" \
24             class="form-control" />
25         <span class="help-block">{{ errors.last_name }}</span>
26     </div>
27 </div>
28 <div class="form-group {%
29 if errors.middle_name %}has-error{%
30 endif %}">
31     <label for="middle_name" class="col-sm-2 control-label">По-батько\ \
32     ві</label>
33     <div class="col-sm-10">
34         <input type="text" value="{{ request.POST.first_name }}" \
35             name="middle_name" id="middle_name" \
36             class="form-control" />
37         <span class="help-block">{{ errors.middle_name }}</span>
38     </div>
39 </div>
40 <div class="form-group {%
41 if errors.birthday %}has-error{%
42 endif %}">
43     <label for="birthday" class="col-sm-2 control-label">Дата Народже\ \
44     ння*</label>
45     <div class="col-sm-10">
46         <input type="text" value="{{ request.POST.birthday }}" \
47             name="birthday" id="birthday" \
48             placeholder="Напр. 1984-12-30" \
49             class="form-control" />
50         <span class="help-block">{{ errors.birthday }}</span>
51     </div>
52 </div>
53 <div class="form-group {%
54 if errors.photo %}has-error{%
55 endif %}">
```

```
52      <label for="photo" class="col-sm-2 control-label">Фото</label>
53      <div class="col-sm-10">
54          <input type="file" value="" name="photo" id="photo"
55              class="form-control" />
56          <span class="help-block">{{ errors.photo }}</span>
57      </div>
58  </div>
59  <div class="form-group { % if errors.ticket %}has-error{ % endif %}">
60      <label for="ticket" class="col-sm-2 control-label">Білет*</label>
61      <div class="col-sm-10">
62          <input type="text" value="{{ request.POST.ticket }}"'
63              name="ticket" id="ticket"
64              class="form-control" />
65          <span class="help-block">{{ errors.ticket }}</span>
66      </div>
67  </div>
68  <div class="form-group { % if errors.student_group %}has-error{ % en\
69 dif %}">
70      <label for="student_group" class="col-sm-2 control-label">Група*<\
71 /label>
72      <div class="col-sm-10">
73          <select name="student_group" id="student_group"
74              class="form-control">
75              <option value="">Виберіть групу</option>
76              { % for group in groups %}
77                  <option value="{{ group.id }}" { % if group.id == request.POS\
78 T.student_group|add:0 %}selected="1"{ % endif %}>{{ group.title }}</o\
79 ption>
80                  { % endfor %}
81          </select>
82          <span class="help-block">{{ errors.student_group }}</span>
83      </div>
84  </div>
85  <div class="form-group { % if errors.notes %}has-error{ % endif %}">
86      <label for="notes" class="col-sm-2 control-label">Додаткові Нота\
```

```
87  тки</label>
88      <div class="col-sm-10">
89          <textarea name="notes" id="notes" class="form-control">{{ requ\ 
90 est.POST.notes }}</textarea>
91          <span class="help-block">{{ errors.notes }}</span>
92      </div>
93  </div>
94  <div class="form-group">
95      <label class="col-sm-2 control-label"></label>
96      <div class="col-sm-10">
97          <input type="submit" value="Додати" name="add_button"
98              class="btn btn-primary" />
99          <button type="submit" name="cancel_button"
100             class="btn btn-link">Скасувати</button>
101     </div>
102 </div>
103 </form>
```

Зверніть увагу на рядок під номером 5. Там ми вставляємо клас “has-error” лише при наявності помилки із полем Ім’я. Подібну річ ми проробили для усіх полів, щоб мати додаткові стилі помилки для тих полів, що не пройшли валідацію даних.

Оновіть сторінку в браузері і перегляньте стилі форми та стилі повідомлень із помилками. Залишилась одна річ: поля форми розтягнуті на усю ширину сторінки. Це може бути непоганим підходом на вужчих екранах, але на широких такі поля виглядають дещо занадто широкими.

Дану проблему ми доволі просто пофіксимо самостійно власним CSS кодом:

Оновляємо стилі нашої форми, файл main.css

```
1 /* Form Styles */  
2 textarea,  
3 select,  
4 input {  
5     max-width: 300px;  
6 }
```

Стилів досить небагато. Все, що ми зробили це: встановили максимальну ширину усіх полів до 300 пікселів. Таким чином ми не обрізали класного функціоналу Twitter Bootstrap, який робить нашу форму гарною на усіх ширинах екранів, а лише обмежили по максимальній ширині.

Знову оновіть сторінку з формою. Ось кінцевий результат наших старань:

Додати Студента

Ім'я*

Прізвище* Введіть ваше прізвище

По-батькові

Дата Народження* Напр. 1984-12-30

Фото Choose File No file chosen

Білет*

Група* Виберіть групу

Додаткові
Нотатки

Форма додавання студента повністю завершена

I не забувайте про регулярні коміти в репозиторій коду.

Гарна робота! Перед тим як переходити до наступної секції пропоную зробити невелику паузу і більше поекспериментувати із матеріалом даної секції. Адже тут ми розібралися із основами роботи із формами. Далі ми з вами продовжимо будувати форми, але з кожною новою секцією даної глави все більше і більше роботи за нас робитиме фреймворк Django.

Форма контакту адміністратора

Ми не одразу стribatимемо від повністю ручної форми додавання студента до повністю автоматичної форми редагування студента, побудованої з допомогою Django форм моделей.

Щоб краще зрозуміти механізм побудови автоматично згенерованої форми, спочатку реалізуємо форму контакту адміністратора сайту. Її ми побудуємо на окремій закладці нашої аплікації “Контакт” і вона дозволить користувачам аплікації надсилати листа адміністратору.

Для цього ми скористаємось Django класом для форми. Цей клас для нас:

- згенерує HTML код полів форми;
- проробить валідацію полів згідно правил, на які ми йому вкажемо.

Нам залишиться лише отримати вже коректні дані і відправити листа поштою.

Підключаємо нову закладку

Почнемо із побудови інфраструктури для нашої нової сторінки із формою контакту.

В urls.py модулі додамо новий шаблон, який обслуговуватиме нашу сторінку:

URL адреса для форми контакту, urls.py модуль

```
1 # Contact Admin Form
2     url(r'^contact-admin/$', 'students.views.contact_admin.contact_a\
3 dmin',
4         name='contact_admin'),
```

Як бачимо із нового URL шаблона, нам потрібно додати новий модуль contact_admin.py в пакеті views аплікації students, а в модулі додати нову функціюв'юшку contact_admin:

Створюємо модуль contact_admin.py

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students/views
2 $ touch contact_admin.py
```

Додаємо базовий код в'юшки, який, поки, буде лише повертати шаблон form.html. Даний шаблон лежатиме в підпапці contact_admin всередині папки із шаблонами:

contact_admin в'юшка

```
1 from django.shortcuts import render
2
3 def contact_admin(request):
4     return render(request, 'contact_admin/form.html', {})
```

Відповідно, тепер маємо створити шаблон form.html, а ще краще, скопіювати його з існуючих шаблонів:

Копіюємо шаблон students/students_add.html в contact_admin/form.html

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students/templates
2
3 $ mkdir contact_admin
4 $ cp students/students_add.html contact_admin/form.html
```

Підредагуємо шаблон form.html і, на даний момент, блок content залишимо практично порожнім:

База для шаблону із формою, form.html

```
1 {% extends "students/base.html" %} 
2
3 {% load static from staticfiles %} 
4
5 {% block meta_title %}Зв'язок із Адміністратором{% endblock meta_title %} 
6
7
8 {% block title %}Зв'язок із Адміністратором{% endblock title %} 
9
10 {% block content %} 
11
12 Тут буде форма контакту. 
13
14 {% endblock content %}
```

На завершення побудови інфраструктури нової сторінки, всередині шаблону base.html, додамо нову закладку Контакт:

Додаємо закладку Контакт

```
1      <ul class="nav nav-tabs" role="tablist">
2          <li role="presentation" {% if request.path == '/' %}class=\
3 "active"{% endif %}><a href="{% url "home" %}">Студенти</a></li>
4          <li role="presentation" {% if '/journal' in request.path %}\
5 }class="active"{% endif %}><a href="/journal">Відвідування</a></li>
6          <li role="presentation" {% if '/groups' in request.path %}\
7 class="active"{% endif %}><a href="{% url "groups" %}">Групи</a></li\
8 >
9          <li role="presentation" {% if '/contact-admin' in request.\
10 path %}class="active"{% endif %}><a href="{% url "contact_admin" %}"\ \
11 >Контакт</a></li>
12      </ul>
```

Зверніть увагу, що в тегу “url” ми скористались іменем URL шаблона, який напередодні прописали в модулі urls.py.

Ніби все. Перевантажуйте сміливо вашу аплікацію в браузері. Повинна з’явитись закладка Контакт, яка покаже порожню сторінку із заголовком: “Зв’язок із Адміністратором”.

Тепер ми готові до реалізації самої форми.

Клас форми

Автоматичні форми в Django будуємо подібно до того, як реалізували класи моделей. Нам потрібно визначити клас нашої форми, унаслідувати від правильної базової класу та набити клас списком необхідних полів.

Автоматичними називаємо ті форми, які генерують HTML код та надають усю валідацію даних без додаткового кастомного коду із сторони розробника.

Ось приклад нашої форми контакту із усіма необхідними для неї полями. Весь код класу форми, як і самої в’юшки, кладемо в contact_admin.py модуль:

Клас форми контакту адміністратора

```
1 from django import forms
2
3 class ContactForm(forms.Form):
4     from_email = forms.EmailField(
5         label=u"Ваша Емейл Адреса")
6
7     subject = forms.CharField(
8         label=u"Заголовок листа",
9         max_length=128)
10
11    message = forms.CharField(
12        label=u"Текст повідомлення",
13        max_length=2560,
14        widget=forms.Textarea)
```

Як бачите ми задекларували клас ContactForm, який унаслідується із базового класу Form. Даний клас приходить із вбудованого в Django пакету: “django.forms”. Все, що пов’язано із формами лежить у даному пакеті.

Далі в класі йде список полів, що нам потрібні на формі контакту:

- емейл адреса відправника;
- заголовок листа;
- та саме тіло повідомлення.

Декларація полів є аналогічною з тим, як ми набивали поля для класів моделей студента та групи. Але в даному випадку ми використовуємо класи полів, що приходять, знову ж таки, із пакету “django.forms”. Ці поля трохи відрізняються від тих, що є моделях. Тут²¹⁹ можна переглянути як співставляються дані поля із полями моделей, а також список усіх доступних полів форм.

²¹⁹<http://djbook.ru/rel1.7/ref/forms/fields.html>

Кожне поле має ряд аргументів для впливу на те, як поле виглядатиме на формі, який тип даних відповідатиме даному полю, яка валідація застосовуватиметься на сервері. Таким чином ми вказали, що хочемо мати Textarea віджет для поля “Текстове повідомлення”. Це для того, щоб користувач міг відправляти цілі параграфи тексту в листі. Також можете бачити, що ми скористались спеціальним полем для емейл адреси відправника. Дане поле за нас зробить валідацію на коректний формат емейл адреси.

Крім генерації HTML коду форми та валідації даних, даний клас форми також перетворить дані, що прийшли від користувача в тілі поста, у Python типи. Таким чином нам не прийдеться робити це самим.

Якщо відчуваєте, що потребуєте трохи більше теоретичної бази по Django формах, тоді рекомендую ознайомитись із [офіційним введенням у форми²²⁰](#).

Логіка відправки листа

Маючи заготовку шаблона та готовий клас форми контакту, можемо приступати до реалізації найцікавішої частини нашого завдання - логіка в'юшки по відправленню листа.

Одразу скажу вам, що коду буде значно менше, ніж ми писали для форми додавання студента, адже усю валідацію та форматування даних для нас зробить клас форми. Ось як виглядатиме кінцевий код модуля contact_admin.py:

Завершена в'юшка відправки емейла адміну

```
1 # -*- coding: utf-8 -*-
2 from django.shortcuts import render
3 from django import forms
4 from django.core.mail import send_mail
5 from django.http import HttpResponseRedirect
6 from django.core.urlresolvers import reverse
7
8 from studentsdb.settings import ADMIN_EMAIL
9
```

²²⁰<http://djbook.ru/rel1.7/topics/forms/index.html>

```
10 class ContactForm(forms.Form):
11     from_email = forms.EmailField(
12         label=u"Ваша Емейл Адреса")
13
14     subject = forms.CharField(
15         label=u"Заголовок листа",
16         max_length=128)
17
18     message = forms.CharField(
19         label=u"Текст повідомлення",
20         widget=forms.Textarea)
21
22 def contact_admin(request):
23     # check if form was posted
24     if request.method == 'POST':
25         # create a form instance and populate it with data from the \
26         request:
27             form = ContactForm(request.POST)
28
29         # check whether user data is valid:
30         if form.is_valid():
31             # send email
32             subject = form.cleaned_data['subject']
33             message = form.cleaned_data['message']
34             from_email = form.cleaned_data['from_email']
35
36             try:
37                 send_mail(subject, message, from_email, [ADMIN_EMAIL\
38             ])
39             except Exception:
40                 message = u'Під час відправки листа виникла непередб\
41                 ачувана ' \
42                         u'помилка. Спробуйте скористатись даною формою п\
43                 ізніше.'
44             else:
```

```
45             message = u'Повідомлення успішно надіслане!'
46
47         # redirect to same contact page with success message
48         return HttpResponseRedirect(
49             u'%s?status_message=%s' % (reverse('contact_admin'), \
50             message))
51
52     # if there was not POST render blank form
53     else:
54         form = ContactForm()
55
56     return render(request, 'contact_admin/form.html', {'form': form})
```

Нас цікавить лише функція в'юшки:

- 24ий рядок: з цим ми уже стикалися раніше; перевіряємо чи форма була відправлена постом;
- 27ий: створюємо об'єкт з класу форми, при цьому передаючи йому POST в якості аргумента; таким чином отримаємо “прив'язану” (англ. bound) форму до даних введених користувачем; це дозволить формі валідувати та конвертувати дані, а пізніше відобразити ці дані на формі при невдалій валідації;
- 30ий: по-справжньому чарівний рядок коду :-); виклик методу `is_valid()` на об'єкті форми заміняє нам усю валідацію даних, а також конвертацію даних у Python типи даних; даний метод повертає `True`, якщо валідація даних пройшла успішно, `False` - у протилежному випадку;
- 32ий: метод `is_valid()` “запихає” усі конвертовані дані з поста запиту в атрибут `“cleaned_data”`; краще використовувати дані із нього, аніж напряму із `“request.POST”`, адже там вони уже правильно підготовані для нашого використання; таким чином отримуємо заголовок листа і зберігаємо його в змінну; аналогічно робимо із рештою полів форми;
- 36ий: розпочинаємо найцікавіше - відправку листа; дана процедура може викинути різноманітні помилки при контакті із сервером, що відправляє листи, тому варто огорнути наступний код гілкою `“try/except”`;

- 37ий: на початку модуля ми імпортнули функцію `send_mail` із пакету “`django.core.mail`”; тепер ми скористаємося даною функцією; вона приймає наступні аргументи в даному порядку: заголовок листа, тіло листа, емейл адреса відправника, список отримувачів; адресу отримувача ми імпортнули із модуля `settings.py` проекту; далі ми налаштуємо емейл сервер, а також додамо дану змінну `ADMIN_EMAIL` до налаштувань; дана функція є надбудовою над Python функцією і полегшує роботу з відправки емейлів; також вона читує налаштування з модуля проекту для нас;
- 39ий: у випадку, якщо функція `send_email` поламалась при спробі відправки листа, ми перехоплюємо усі помилки і встановлюємо відповідне статусне повідомлення про поломку; дане повідомлення ми далі використаємо при редіректі;
- 44ий: якщо ж відправка відбулась успішно (так, “`try/except`” мають ще й гілку “`else`”), ми також встановлюємо статусне повідомлення, але тепер про позитивний результат відправки листа;
- 48ий: незалежно від результату ми редіректимо назад на форму контакту, при цьому показуючи статусне повідомлення встановлене раніше;
- 53ій: починаємо гілку коду, коли поста форми не було;
- 54ий: в такому випадку ми також створюємо об’єкт з класу форми, але цього разу не передаємо йому даних із запиту; таким чином форма буде порожньою на сторінці;
- 56ий: віддаємо шаблон сторінки, передаючи об’єкт форми.

Зауважте: якщо форма буде запощена, але дані не будуть проходити валідацію, тоді в шаблон ми передамо “прив’язану” форму (тобто форму з даними від попереднього запиту). Таким чином, при некоректно введених даних, користувач отримає заповнені поля і матиме змогу виправити помилки без повторного введення усіх полів.

Тепер, коли уся логіка функції в’юшки на місці, маємо зробити ще дві речі, щоб заставити це все запрацювати:

- налаштувати сервер відправки емейлів;
- оновити шаблон, щоб використовував передану йому форму, а також правильно відображав статусні повідомлення.

SMTP сервер відправки листів

Дана секції не включає детальне налаштування самого SMTP сервера (сервера з відправки листів), а лише надає огляд кількох варіантів відправки листів. Тут ми сфокусуємося над тим, як і де правильно прописати деталі вашого підготовленого сервера в коді нашого Django проекту.

Завдання щодо налаштування власного SMTP сервера залишається вам на самостійне опрацювання.

Функція `send_mail`, яку ми використовуємо для відправки листів, потребує кількох налаштованих змінних в модулі `settings.py` проекту:

- `EMAIL_HOST`: адреса вашого SMTP сервера;
- `EMAIL_PORT`: порт SMTP сервера;
- `EMAIL_HOST_USER`: ім'я користувача, якщо доступ до сервера закритий;
- `EMAIL_HOST_PASSWORD`: пароль користувача, якщо доступ до сервера закритий;
- `EMAIL_USE_TLS`: чи використовувати безпечне TLS з'єднання під час контакту SMTP сервера; це залежить від типу сервера і його налаштувань;
- `EMAIL_USE_SSL`: взаємовиключна опція із `EMAIL_USE_TLS`; якщо одна з них включена, тоді інша обов'язково має бути виключена; якщо не впевнені, який саме режим безпеки працює на вашому SMTP сервері - пробуйте різні комбінації перемикання даних опцій, а також варіант, коли обидві виключені.

Також в `settings.py` потрібно додати змінну `ADMIN_EMAIL`, яка вказуватиме на емейл адресу адміністратора нашої веб-аплікації:

Приклад налаштувань SMTP сервера

```
1 # email settings
2 # please, set here you smtp server details and your admin email
3 ADMIN_EMAIL = 'admin@studentsdb.com'
4 EMAIL_HOST = 'smtp.server.com'
5 EMAIL_PORT = '587'
6 EMAIL_HOST_USER = 'username1'
7 EMAIL_HOST_PASSWORD = '*****'
8 EMAIL_USE_TLS = True
9 EMAIL_USE_SSL = False
```

Взалежності від того, як налаштуєте ваш SMTP сервер (або яким зовнішнім сервером скористаєтесь), відповідно прийдеться оновити значення даних змінних.

Є як мінімум три варіанта для відправки листів:

Скористатись SMTP сервером вашого поштовика

Якщо у вас, наприклад, є екаунт на gmail.com, тоді ви можете скористатись ним для відправки листів із нашої аплікації.

На [даній сторінці²²¹](#) ви можете знайти деталі доступу і використання Gmail SMTP сервера, а також інших популярних емейл сервісів.

Ось приклад моїх налаштувань у випадку використання Gmail сервера для відправки листів:

²²¹<http://ru.email-unlimited.com/help/mailersmtp-settings-samples.html>

Приклад налаштувань для Gmail SMTP сервера

```
1 # email settings
2 # please, set here you smtp server details and your admin email
3 ADMIN_EMAIL = 'admin@studentsdb.com'
4 EMAIL_HOST = 'smtp.gmail.com'
5 EMAIL_PORT = '465'
6 EMAIL_HOST_USER = 'vitaliyopodoba@gmail.com'
7 EMAIL_HOST_PASSWORD = '*****'
8 EMAIL_USE_TLS = False
9 EMAIL_USE_SSL = True
```

Спеціалізований сервіс відправки листів

Також існує маса зовнішніх сервісів для відправки листів. Вони надають SMTP сервер для використання і, так само як і з вашим поштовиком, вам залишається лише додати деталі сервера в модуль налаштування проекту.

Я рекомендую оглянути наступні сервіси:

- <https://mandrillapp.com> - користуюсь ним найбільше в даний момент;
- <https://sendgrid.com/> - один із найпопулярніших сервісів розсилки листів на даний момент;
- <http://www.icontact.com/> - ще один непоганий сервіс відправки листів.

Зареєструйтесь на одному із них. Відправка невеликої кількості емейлів зазвичай є безкоштовною. Для тестування точно вистачить. Після реєстрації налаштуйте сервіс та отримайте дані доступу до його SMTP серверу.

Ось як виглядатиме приклад моїх налаштувань через сервіс Mandrill:

Приклад налаштувань для Mandrill SMTP сервера

```
1 # email settings
2 # please, set here you smtp server details and your admin email
3 ADMIN_EMAIL = 'admin@studentsdb.com'
4 EMAIL_HOST = 'smtp.mandrillapp.com'
5 EMAIL_PORT = '587'
6 EMAIL_HOST_USER = 'vitaliyopodoba@gmail.com'
7 EMAIL_HOST_PASSWORD = '*****'
8 EMAIL_USE_TLS = True
9 EMAIL_USE_SSL = False
```

Власний SMTP сервер

Вкінці-кінців можете спробувати встановити і налаштувати власний SMTP сервер на локальній машині.

На Лінуксі одними із найпопулярніших є [sendmail²²²](#) та [postfix²²³](#).

Якщо дійсно хочете спробувати даний варіант, тоді інсталяція та конфігурація даних серверів залишається вам повністю на самостійне опрацювання.

Після успішного налаштування власного SMTP сервера ваш `settings.py` модуль повинен містити щось подібне на:

Приклад налаштувань локального SMTP сервера

```
1 # email settings
2 # please, set here you smtp server details and your admin email
3 ADMIN_EMAIL = 'admin@studentsdb.com'
4 EMAIL_HOST = 'localhost'
5 EMAIL_PORT = '25'
6 EMAIL_HOST_USER = ''
7 EMAIL_HOST_PASSWORD = ''
8 EMAIL_USE_TLS = False
9 EMAIL_USE_SSL = False
```

²²²<http://uk.wikipedia.org/wiki/Sendmail>

²²³<http://uk.wikipedia.org/wiki/Postfix>

Думаю ви не будете налаштовувати ні доступ по користувачу, ні безпечні протоколи.

...

Тут²²⁴ можете знайти більше деталей про конфігурацію налаштувань для сервера відправки листів.

А на [даній сторінці](#)²²⁵ є непоганий вступ по роботі із листами в Django.

Якщо все зробили правильно, то за 30 хвилин матимете код по відпраці листа готовий.

Оновлюємо шаблон форми

Залишилось зовсім небагато, щоб закінчити із відправкою листів: оновити шаблон, де скористатись формою, яку нам передає в'юшка.

Нам потрібно оновити два блоки: `content` та `status_message`. І почнемо ми із важливішого - блоку `content`.

Об'єкт форми надає нам кілька корисних методів для генерації HTML коду форми:

- `{{ form.as_p }}`: огортає поля (мітку + поле вводу) в теги параграфів (`p`);
- `{{ form.as_table }}`: представляє форму у вигляді таблиці;
- `{{ form.as_ul }}`: представляє поля у вигляді списку (теги “`ul/li`”);
- ну і на кінець: якщо вставити об'єкт форми в шаблон без використання вищеперечислених методів, тоді теги полів не будуть огорнуті в жодні додаткові теги.

Ще хочу звернути вашу увагу на те, що форма не згенерує для нас самого тегу форми та кнопок, а лише HTML код для списку полів задекларованих в класі форми. Так само тег CSRF захисту потрібно вставити самостійно. Ось кінцевий результат блоку `content`:

²²⁴<http://djbook.ru/rel1.7/ref/settings.html#email-host>

²²⁵<http://djbook.ru/rel1.7/topics/email.html>

Додаємо форму контакту, contact_admin.form.html

```
1  {% block content %}

2
3  <form action="{% url "contact_admin" %}" method="post">
4      {% csrf_token %}
5
6      {{ form.as_p }}
7
8      <input type="submit" value="Надіслати" name="send_button" />
9  </form>
10
11  {% endblock content %}
```

Метод запиту форми POST, а зсилається форма сама на себе, адже наша в'юшка займається як генерацією форми, так і її обробкою.

Все решта, думаю, для вас є зрозумілім у вищеннаведеному прикладі.

Уже на даний момент форма має бути повністю робочою. Статусні повідомлення щодо успішної відправки та поломки під час спроби відправити листа також уже мають працювати. Спробуйте потестувати функціонал форми. Якщо отримуєте помилку при спробі відправити валідні дані, повертайтесь у секцію з налаштування SMTP сервера і спробуйте полагодити значення змінних в модулі `settings.py`.

Але маємо ще один боржок щодо функціоналу форми. У випадку, якщо дані не пройшли валідацію, ми отримуємо помилки зверху некоректно введених полів, але немає глобального статусного повідомлення.

Для цього нам потрібно повністю перекрити блок `status_message`, що приходить із шаблону `base.html` під потреби нашої форми:

Перекриваємо блок status_message

```
1  {% block status_message %}  
2  
3  {% if form.errors %}  
4      <div class="alert alert-warning" role="alert">  
5          Будь-ласка, виправте наступні помилки  
6      </div>  
7  {% endif %}  
8  
9  {{ block.super }}  
10  
11  {% endblock %}
```

Що ми зробили?

- Зій рядок: усі назбирани помилки під час валідації даних від користувача форма для нас збирає в атрибут errors; тому ми скористалися ним, щоб перевірити чи форма пройшла валідацію;
- 4ий: якщо є помилки, тобто валідація не пройшла, виводимо статичне повідомлення і просимо користувача виправити їх;
- 9ий: block.super - цікавий момент; нам потрібен кусок коду, який виводитиме статусні повідомлення, які ми передаємо під час редіректу в URL параметрі status_message; і щоб не дублювати коду із base.html, ми вставляємо його в кінцевий шаблон через “block.super”; атрибут “super” вказує на батьківський блок, який у нашому випадку є блоком із шаблону base.html.

Зверніть увагу на те, що автоматичні повідомлення про помилки на формі є уже для нас переведеними на українську мову. Це завдяки прописаній змінній “LANGUAGE_CODE = ‘uk’” в модулі settings.py. Django форми уже мають український переклад інтерфейсу форм вкладений по-замовчуванню. Пізні-

ше ми повернемось до цієї теми і повністю перекладемо нашу аплікацію на українську мову правильним чином.

Оновіть форму і ще раз потестуйте її. Весь функціонал на місці! Ось як виглядатиме ваша форма у випадку, якщо відбудеться помилка при валідації даних:

The screenshot shows a contact form titled "Зв'язок із Адміністратором". At the top, there are tabs: "Студенти", "Відвідування", "Групи", and "Контакт". The "Контакт" tab is active. A yellow error message box contains the text: "Будь-ласка, виправте наступні помилки". Below this, the form fields are displayed:

- "Ваша Емейл Адреса": vitaliyopodoba@gmail.com (highlighted in red)
- A red asterisk next to the email field indicates it is required.
- "Заголовок листа": [empty input field]
- "Текст повідомлення": [text area containing "Тестове повідомлення"]

At the bottom of the form are two buttons: "Надіслати" (Send) and a copyright notice: "© 2014 Сервіс Обліку Студентів".

Форма контакту адміна практично готова

Єдине, що залишилось це додати гарні стилі.

Стилі Twitter Bootstrap через Crispy Forms аплікацію

Знову залишився лише зовнішній вигляд форми. А саме:

- підрівняти мітки і поля;
- застилити стани із помилками в полях;
- застилити гарно кнопку.

Стилі знову ж таки візьмемо із бібліотеки Twitter Bootstrap. Але цього разу ми маємо далеко не повний контроль над HTML кодом форми. Відповідно потрібен інший підхід для стилізації.

Скористаємося однією з існуючих Django аплікацій, які надписують HTML стандартних Django форм кодом, що підходить під Twitter Bootstrap правила. Хорошим балансом між функціоналом та простотою у використанні володіє аплікація [Crispy Forms²²⁶](#). Нею ми і скористаємося, щоб “облагородити” нашу форму контакту.

Згідно [інсталяційної документації²²⁷](#) пакету під’єднаємо Crispy Forms до нашого проекту. Будемо використовувати останню стабільну версію - 1.4.0.Хоча вона і не згадана в документації, проте я знайшов її серед тегів в [репозиторії коду²²⁸](#). Відповідно, частину речей вам прийдеться вичитувати з коду, оскільки дещо може бути не включено у застарілу документацію аплікації.

Додамо пакет версії 1.4.0 в requirements.txt та заінсталюємо його у наше робоче середовище:

requirements.txt

```
1 Django==1.7.1
2 MySQL-python
3 Pillow
4 django-crispy-forms==1.4.0
```

Запускаємо інсталяцію пакетів

```
1 # не забудьте перед даною командою активувати своє
2 # віртуальне середовище
3 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb
4 $ pip install -r requirements.txt
```

Додаємо пакет crispy_forms у список заінсталюваних аплікацій:

²²⁶<https://readthedocs.org/projects/django-crispy-forms/>

²²⁷<http://django-crispy-forms.readthedocs.org/en/1.2.1/install.html>

²²⁸<https://github.com/maraujop/django-crispy-forms/tree/1.4.0>

Додаємо crispy_forms до INSTALLED_APPS всередині settings.py модуля

```
1 INSTALLED_APPS = (
2     'django.contrib.admin',
3     'django.contrib.auth',
4     'django.contrib.contenttypes',
5     'django.contrib.sessions',
6     'django.contrib.messages',
7     'django.contrib.staticfiles',
8     'crispy_forms',
9     'students',
10 )
```

Конфігуруємо налаштування нової аплікації:

settings.py

```
1 CRISPY_TEMPLATE_PACK = 'bootstrap3'
```

Змінна “CRISPY_TEMPLATE_PACK” вказує аплікації Crispy Forms, які саме шаблони використовувати, адже вона містить не один набір для різних фреймворків. Нам, звичайно, потрібен останній Twitter Bootstrap: “bootstrap3”.

Підключати CSS і Javascript файли не будемо, адже вони у нас вже давно підключені.

Тепер ми дійшли до найцікавішої частини - будемо додавати стилі до форми. Але будемо це робити у трохи незвичний спосіб: через Python код в класі форми, а не у шаблоні чи main.css файлі.

Crispy Forms дає нам два типи об'єктів для впливу на вигляд форми:

- **FormHelper клас²²⁹:** власне через нього ми будемо робити усі наші зміни; його ми створимо в __init__.py методі класу форми;

²²⁹http://django-crispy-forms.readthedocs.org/en/latest/form_helper.html

- **лайаути**²³⁰: вони дозволяють змінювати теги та поля на формі, а також, наприклад, додавати кнопки на форму (адже по-замовчуванню форма буде без кнопок); нам буде потрібен лише один лейаут - лейаут кнопки.

Ось як виглядатиме оновлений клас форми:

Клас форми із crispy forms FormHelper об'єктом

```

1 from crispy_forms.helper import FormHelper
2 from crispy_forms.layout import Submit
3
4 class ContactForm(forms.Form):
5
6     def __init__(self, *args, **kwargs):
7         # call original initializer
8         super(ContactForm, self).__init__(*args, **kwargs)
9
10        # this helper object allows us to customize form
11        self.helper = FormHelper()
12
13        # form tag attributes
14        self.helper.form_class = 'form-horizontal'
15        self.helper.form_method = 'post'
16        self.helper.form_action = reverse('contact_admin')
17
18        # twitter bootstrap styles
19        self.helper.help_text_inline = True
20        self.helper.html5_required = True
21        self.helper.label_class = 'col-sm-2 control-label'
22        self.helper.field_class = 'col-sm-10'
23
24        # form buttons
25        self.helper.add_input(Submit('send_button', 'Надіслати'))

```

²³⁰<http://django-crispy-forms.readthedocs.org/en/latest/layouts.html>

У даному прикладі ми не наводили списку полів, адже вони залишились незмінними.

Давайте детальніше по-рядках проаналізуємо метод `__init__.py`:

- 1ий рядок: імпортуємо необхідний мінімум класів із crispy_forms аплікації; `Submit` - лейаут кнопки, яким ми далі в коді скористаємося;
- 11ий: створюємо об'єкт типу `FormHelper` і запам'ятовуємо його в атрибуті “`helper`” об'єкту форми; важливо, щоб атрибут мав саме дану назву, тоді Crispy Forms шаблонний тег підхопить нашого хелпера автоматично;
- 13ий: блок коду, де ми модифікуємо сам тег форми, встановлюючи йому клас, метод та атрибут ‘`action`’; так, Crispy Forms згенерує для нас повну форму, включно з тегом `form`, CSRF захистом та кнопками; пізніше ми оновимо відповідно наш шаблон;
- 18ий: блок коду, де ми встановлюємо специфічні атрибути для Twitter Bootstrap бібліотеки, а саме: включаємо HTML5 атрибути валідації та додаємо класи Grid лейауту для мітки та поля, щоб краще розташувати елементи форми на сторінці;
- 25ий: додаємо з коду лейаут кнопки `Submit`; ця стрічка коду додасть нам кнопку “Надіслати” до форми із назвою “`send_button`”.

Все, що потрібно зробити із змінами у формі, ми зробили. Тепер потрібно оновити наш шаблон, щоб він показував не стандартну, а кастомізовану форму, яка приходить із Crispy Forms HTML кодом.

Саме для цього дана аплікація надає тег “`{% crispy [form_name] %}`”, якому потрібно передати об'єкт форми. Даний тег згенерує повний тег форми із усіма змінами, які ми занотували в Python коді.

Ось як доволі просто виглядатиме кінцевий шаблон:

Шаблон форми використовує Crispy Forms

```
1  {% extends "students/base.html" %}  
2  
3  {% load static from staticfiles %}  
4  {% load crispy_forms_tags %}  
5  
6  {% block meta_title %}Зв'язок із Адміністратором{% endblock meta_tit\le %}  
7  
8  
9  {% block title %}Зв'язок із Адміністратором{% endblock title %}  
10  
11  {% block status_message %}  
12  {% if form.errors %}  
13      <div class="alert alert-warning" role="alert">  
14          Будь-ласка, виправте наступні помилки  
15      </div>  
16  {% endif %}  
17  
18  {{ block.super }}  
19  {% endblock %}  
20  
21  
22  {% block content %}  
23  
24  {% crispy form %}  
25  
26  {% endblock content %}
```

Що ми змінили:

- 4ий рядок: завантажили теги з Crispy Forms аплікації; по-замовчуванню зовнішні аплікації не підключають свої теги в шаблони;
- 24ий: викликали тег “{% crispy %}” і передали йому нашу форму.

Так просто ми отримали кінцеву форму. Тепер перезавантажте сторінку із формою і гляньте на результат:

Зв'язок із Адміністратором

Ваша Емейл
Адреса*

Заголовок листа*

Текст
повідомлення*

Надіслати

Форма контакту адміністратора із Crispy Forms

Як бачите, форма уже виглядає значно краще. Crispy Forms аплікація на всі 100% задовольнила наші потреби.

На домашнє завдання: переробіть контакт форму з використанням вже існуючої Django аплікації: <https://pypi.python.org/pypi/django-contact-form> Попуті, вам практично не прийдеться розробляти кастомного коду, а лише інтегрувати існуючий у свою аплікацію.

...

Тепер ми не лише вмімо з нуля створювати форму та працювати з нею, але й знаємо швидший шлях для отримання даних від користувача - Django форми. Будемо ще більше прискорюватись! Переходимо до Django форм створених спеціально для роботи із моделями.

Форма редагування студента

Форма редагування, мабуть, буде найшвидшою формою, яку ми реалізуємо. Без написання HTML коду полів, практично без Python логіки, декларації полів і навіть класу форми! Все це за нас зробить форма Django для роботи з моделями.

Для в'юшки форми редагування студента ми вперше скористаємось не функцією, а класом. Тому давайте спочатку коротко оглянемо різницю між цими двома підходами, а також коли використовувати кожен із них.

В'юшки Функції vs В'юшки Класи

В'юшки Класи ввели в Django відносно недавно. До цього ми могли створювати в'юшки лише використовуючи Python функції.

Основною задачею класів є зменшити дубльований код між в'юшками. Python класи дозволяють визначати абстрактні класи, в яких міститься повторюваний код, і потім унаслідуватись від цих класів у кінцевих в'юшках.

Програмування веб-сайтів це, досить часто, багато повторюваної роботи. Якраз таку повторювану роботу і спробували винести у наперед заготовані класи в'юшок в Django. Таким чином, якщо ваше завдання підпадає під категорію таких повторюваних рутин, тоді ви зможете зекономити масу часу, скориставшись готовим класом.

Ось список визначених для нас класів в Django:

- ListView: для виведення списку об'єктів з бази;
- DetailView: для відображення одного об'єкту з бази;
- RedirectView: для здійснення редіректу на іншу сторінку; так, навіть для цієї дії є окрема в'юшка-клас;
- CreateView: форма додавання нового об'єкта до бази;
- UpdateView: форма редагування існуючого об'єкта в базі;
- DeleteView: видалення об'єкта з бази;

- ArchiveIndexView, YearArchiveView, MonthArchiveView, WeekArchiveView, DayArchiveView, TodayArchiveView, DateDetailView: в'юшки для реалізації сторінок блогу.

Усі вищеперечислені класи певним чином використовують один із базових класів: FormView, View, TemplateView. До речі, класом TemplateView ми самі скористаємось під час реалізації закладки Відвідування.

Клас чи Функція?

Є три табора розробників у спільноті Django:

- ті, хто до останнього уникає класів; тобто завжди починають із функції, а далі переходятъ на клас, якщо вже немає куди діватись;
- ті, хто намагаються всюди “втулити” клас;
- ті, хто аналізує ситуацію, завдання і проект загалом, та кожного разу вирішує, що краще використати.

Я, особисто, надаю перевагу третьому табору, і кожного разу обираю той підхід, який краще пасуватиме під конкретну задачу. Ось кілька критеріїв, якими я керуюсь при цьому:

Якщо ваше завдання підпадає під стандартні задачі:

- відобразити об'єкт чи список об'єктів з бази даних;
- додати, поредагувати чи видалити об'єкт з бази;
- написати блог,

тоді використовуйте наперед заготовані в Django класи в'юшок.

Якщо ви повторюєте багато коду у різних функціях в'юшок в проекті, тоді виділіть цей код в базовий абстрактний клас, переробіть функції на класи і унаслідуйтесь від базового класу.

Якщо завдання ще не до кінця вам зрозуміле, тоді використовуйте функцію. Пізніше, при потребі, переїдете на клас.

Якщо маєте в'юшку у вигляді класу, але з часом кастомізувати її під потреби проекту стає все важче і важче. Тоді є зміст задуматись над поверненням на функцію, де ви будете вільні від будь-яких правил.

Ось і всі правила, якими я зазвичай керуюсь при виборі між класом та функцією для чергової в'юшки. З досвідом ваш вибір ставатиме швидшим і кращим.

Робота із класами

Робота із класами в якості в'юшок дещо відрізняється від того, як ми працювали із функціями. Класи дають нам багато наперед заготовленого функціоналу, але і разом з тим багато обмежень і правил, які нам потрібно знати. Почнемо із огляду класів для відображення даних на сторінці, а потім перейдемо до класів роботи із формами.

При кожному запиті запускається функція в'юшки. А коли маємо справу із класами, тоді при кожному запиті створюється об'єкт із класу в'юшки і запускається її вбудований метод `dispatch231`. Цей метод унаслідуємо від вбудованих Django в'юшок. Даний метод запускає одноіменний метод з назвою типу запиту: `get`, `post`, `head` і т.д. Дані методи об'єкта класу можемо як використовувати по-замовчуванню із унаслідуваної базової в'юшки, так і описувати у власному класі при потребі.

Таким чином, для кожного аспекта обробки запиту, даних і формування відповіді, в Django класах в'юшок передбачені свої атрибути та методи класу. Потрібно знати набір найбільш популярних методів та атрибутів, щоб мати можливість їх кастомізувати на рівні власного класу.

Найбазовішим класом є `View232` і він має такі важливі для нас методи та атрибути:

- `as_view233`: статичний метод²³⁴ (має декоратор `staticmethod`) використовується в URL шаблонах, адже ми не можемо напряму передати клас в `url` функцію в модулі `urls.py`;

²³¹<http://djbook.ru/rel1.7/ref/class-based-views/base.html#django.views.generic.base.View.dispatch>

²³²<http://djbook.ru/rel1.7/ref/class-based-views/base.html#django.views.generic.base.View>

²³³http://djbook.ru/rel1.7/ref/class-based-views/base.html#django.views.generic.base.View.as_view

²³⁴<https://docs.python.org/2/library/functions.html#staticmethod>

- get, post, head і т.д.: для обробки запиту відповідного типу.

Клас [TemplateView²³⁵](#) додатково підготує і віддасть у відповіді шаблон з HTML кодом:

- template_name: атрибут, що вказує на шлях до файлу шаблона;
- [get_context_data²³⁶](#): метод повертає словник із додатковими даними для шаблона.

Клас [RedirectView²³⁷](#):

- [get_redirect_url²³⁸](#): метод, що віддає URL адресу для редіректу;
- url: атрибут, в якому можна містити статичну стрічку-адресу для редіректу.

Клас [DetailView²³⁹](#) працює для генерації списку об'єктів із бази даних:

- [model²⁴⁰](#): атрибут, який вказує на клас моделі, з яким працюватиме дана в'юшка;
- [get_queryset²⁴¹](#): метод, який задає кастомний запит в базу для отримання одного об'єкта заданого типу;
- get_context_data: такий самий метод, як і в TemplateView класі;
- get_object: повертає об'єкт із бази, який будемо представляти на сторінці.
- [render_to_response²⁴²](#)

²³⁵<http://djbook.ru/rel1.7/ref/class-based-views/base.html#templateview>

²³⁶http://djbook.ru/rel1.7/ref/class-based-views/mixins-simple.html#django.views.generic.base.ContextMixin.get_context_data

²³⁷<http://djbook.ru/rel1.7/ref/class-based-views/base.html#redirectview>

²³⁸http://djbook.ru/rel1.7/ref/class-based-views/base.html#django.views.generic.base.RedirectView.get_redirect_url

²³⁹<http://djbook.ru/rel1.7/ref/class-based-views/generic-display.html#detailview>

²⁴⁰<http://djbook.ru/rel1.7/ref/class-based-views/mixins-multiple-object.html#django.views.generic.list.MultipleObjectMixin.model>

²⁴¹http://djbook.ru/rel1.7/ref/class-based-views/mixins-single-object.html#django.views.generic.detail.SingleObjectMixin.get_queryset

²⁴²http://djbook.ru/rel1.7/ref/class-based-views/mixins-simple.html#django.views.generic.base.TemplateResponseMixin.render_to_response

- **object**: змінна доступна в шаблоні для отримання об'єкта, який відображаємо.

Клас `ListView`²⁴³, який використовуємо для відображення списку об'єктів із бази даних:

- `paginate_by`²⁴⁴: атрибут, який вказує на кількість елементів на сторінці; так, даний клас реалізує для нас посторінкову навігацію;
- `object_list`²⁴⁵: список об'єктів під даною назвою доступний в шаблоні;
- також даний клас передає шаблону змінні для роботи із посторінковою навігацією по списку: `is_paginated`, `paginator`, `page_obj`;
- усі інші методи подібні як у класі `DetailView`: `get_context_data`, `get_queryset`.

...

Давайте тепер розглянемо невеличкий, швидше теоретичний, приклад класу в'юшки і його використання. Зробимо в'юшку списку студентів, але цього разу використовуючи класи. Скористаємось базовим класом `ListView`:

Клас в'юшки списку студентів

```

1 from django.views.generic import ListView
2 from students.models.students import Student
3
4 class StudentList(ListView):
5     model = Student

```

Додамо дану в'юшку до URL шаблонів:

²⁴³<http://djbook.ru/rel1.7/ref/class-based-views/generic-display.html#django.views.generic.list.ListView>

²⁴⁴http://djbook.ru/rel1.7/ref/class-based-views/mixins-multiple-object.html#django.views.generic.list.MultipleObjectMixin.paginate_by

²⁴⁵http://djbook.ru/rel1.7/ref/class-based-views/mixins-multiple-object.html#django.views.generic.list.MultipleObjectMixin.get_context_data

Ось як лінкуються класові в'юшки в urls.py

```
1 from django.conf.urls import patterns, url
2 from students.views.students import StudentList
3
4 urlpatterns = patterns('',
5     url(r'^student_list/$', StudentList.as_view()),
6 )
```

І це увесь Python код, який нам потрібно писати для списку студентів. Залишився шаблон. Зробимо його дуже простим:

Шаблон для списку студентів

```
1 {% extends "base.html" %}
2
3 {% block content %}
4     <h2>Студенти</h2>
5     <ul>
6         {% for student in object_list %}
7             <li>{{ student.last_name }}</li>
8         {% endfor %}
9     </ul>
10    {% endblock %}
```

Клас в'юшки автоматично шукатиме за шаблоном із назвою “students/student_list.html”, де “students” береться із назви аплікації, “student” - із назви моделі, “list” - із назви в'юшки “ListView”. Або, альтернативно, можемо додати атрибут класу “template_name” до класу StudentList, щоб використати шаблон із іншою назвою.

В шаблоні ми скористались змінною “object_list”, який нам передала в'юшка. Це є список усіх студентів в базі. Ми можемо обмежити даний список через get_queryset метод на класі в'юшки, а також можемо встановити додаткові дані для шаблону і ще декілька додаткових кастомізацій:

Більше кастомізацій у класі в'юшки

```
1 from django.views.generic import ListView
2 from students.models.students import Student
3
4 class StudentList(ListView):
5     model = Student
6     context_object_name = 'students'
7     template = 'students/student_class_based_view_template'
8
9     def get_context_data(self, **kwargs):
10         """This method adds extra variables to template"""
11         # get original context data from parent class
12         context = super(StudentList, self).get_context_data(**kwargs)
13
14         # tell template not to show logo on a page
15         context['show_logo'] = False
16
17         # return context mapping
18         return context
19
20     def get_queryset(self):
21         """Order students by last_name."""
22         # get original query set
23         qs = super(StudentList, self).get_queryset()
24
25         # order by last name
26         return qs.order_by('last_name')
```

В ускладненому класі ми:

- змінили називу змінної, що містить список студентів в шаблоні на “students”;
- встановили кастомну називу шаблона через атрибут “template”;
- додали змінну “show_logo” в контекст шаблона;
- змінили порядок студентів посортувавши їх по прізвищу.

Класи для роботи із формами

На завершення теоретичної частини про в'юшки-класи, розглянемо ще набір визначених для нас класів по роботі із формами.

Базовий клас для усіх в'юшок із формами називається [FormView²⁴⁶](#). Він відображає форму, показує помилки валідації і при успішній дії - редіректить на нову URL адресу. Ось основні атрибути і методи:

- подібні до атрибутів і методів, які ми уже з вами розглянули у попередніх класах: template_name, get, post, put, dispatch, as_view, get_context_data, model;
- fields: атрибут; вказує на список полів, які відобразити на формі;
- [form_class²⁴⁷](#): атрибут, який вказує на клас форми; замість нього можна визначати метод [get_form_class²⁴⁸](#);
- initial: атрибут, словник із початковими даними для форми;
- success_url: атрибут, адреса сторінки для редіректу після успішної обробки форми; замість нього можна використовувати метод [get_success_url²⁴⁹](#);
- form_valid: метод, який редіректить на сторінку при успішній обробці форми;
- form_invalid: рендерить відповідь користувачу, що містить форму із вказаними помилками.

Клас для форми створення нового об'єкта [CreateView²⁵⁰](#):

- object: атрибут, який вказує на новостворений об'єкт;
- решта методів і атрибутів, що були згадані в переліку до класу FormView.

²⁴⁶ <http://djbook.ru/rel1.7/ref/class-based-views/generic-editing.html#django.views.generic.edit.FormView>

²⁴⁷ http://djbook.ru/rel1.7/ref/class-based-views/mixins-editing.html#django.views.generic.edit.FormMixin.form_class

²⁴⁸ http://djbook.ru/rel1.7/ref/class-based-views/mixins-editing.html#django.views.generic.edit.FormMixin.get_form_class

²⁴⁹ http://djbook.ru/rel1.7/ref/class-based-views/mixins-editing.html#django.views.generic.edit.FormMixin.get_success_url

²⁵⁰ <http://djbook.ru/rel1.7/ref/class-based-views/generic-editing.html#createview>

Клас для форми редагування існуючого об'єкта `UpdateView`²⁵¹. Підтримує усі атрибути і методи визначені у класі `CreateView`.

Клас для видалення існюючого об'єкта із бази `DeleteView`²⁵². Найважливіші атрибути - це `model` та `success_url`.

Як користуватись базовою формою і в'юшкою-функцією ми уже знаємо із секції про розробку форми контакту адміністратора. А ось так можна це зробити використовуючи клас. Наводжу спрощений варіант логіки:

Контакт форма через клас в'юшки

```
1 from django.views.generic.edit import FormView
2
3 class ContactView(FormView):
4     template_name = 'contact_form.html'
5     form_class = ContactForm
6     success_url = '/email-sent/'
7
8     def form_valid(self, form):
9         """This method is called for valid data"""
10        subject = form.cleaned_data['subject']
11        message = form.cleaned_data['message']
12        from_email = form.cleaned_data['from_email']
13
14        send_mail(subject, message, from_email, ['admin@gmail.com'])
15
16        return super(ContactView, self).form_valid(form)
```

На домашнє завдання: повністю перепишіть функцію-в'юшку форми контакту адміністратора на клас, який буде унаслідуватись від `FormView`.

²⁵¹<http://djbook.ru/rel1.7/ref/class-based-views/generic-editing.html#updateview>

²⁵²<http://djbook.ru/rel1.7/ref/class-based-views/generic-editing.html#deleteview>

FormView клас для нас визначає набір методів і атрибутів таким чином, що нам не треба писати багато умовних операторів для перевірки на тип запиту і тому подібні речі.

Форми роботи з моделями

Класи UpdateView, CreateView і DeleteView є, так званими, в'юшками для роботи із моделями. Кожна із них змінює певним чином базу даних і операє над тим чи іншим класом моделі.

Щоб вказати даним класам, з якими об'єктами вони працюватимуть використовуються наступні варіанти в порядку їхньої важливості:

- якщо визначений атрибут “model” на рівні класу, тоді він використовується;
- якщо визначений метод “get_object” повертає об'єкт, тоді клас даного об'єкту використовуватиметься як модель;
- якщо визначений “queryset”, тоді модель визначена у ньому буде використовуватись у подальшій роботі класу-в'юшки.

...

На цьому з теорією завершуємо і переходимо до практики. Реалізуємо форму редагування студента.

Але вам, перед переходом до практики, рекомендую ще добряче начитатись про форми²⁵³ і спробувати самостійно розібратись, якщо інформації даної секції не вистачило вам для повного розуміння.

В'юшка-Клас для редагування студента

Цього разу почнемо не із під'єднання шаблонів, а прямо із класу в'юшки. Відкриваємо у своєму улюбленому редакторі коду модуль students.py із пакету views в аплікації students і додаємо наступний клас:

²⁵³<http://djbook.ru/rel1.7/topics/class-based-views/index.html>

Клас-в'юшка для редагування студента

```
1 from django.http import HttpResponseRedirect, HttpResponseRedirect
2 from django.core.urlresolvers import reverse
3 from django.views.generic import UpdateView
4
5 from ..models import Student, Group
6
7 class StudentUpdateView(UpdateView):
8     model = Student
9     template_name = 'students/students_edit.html'
10
11     def get_success_url(self):
12         return u'%s?status_message=Студента успішно збережено!' \
13             % reverse('home')
14
15     def post(self, request, *args, **kwargs):
16         if request.POST.get('cancel_button'):
17             return HttpResponseRedirect(
18                 u'%s?status_message=Редагування студента відмінено!' \
19                 %
20                     reverse('home'))
21         else:
22             return super(StudentUpdateView, self).post(request, *arg\
23 s, **kwargs)
```

Думаю, більшість речей зрозуміла базуючись на матеріалі попередньої секції, але, все ж таки, коротенько пройдемось по важливих моментах:

- Зій рядок: усі дефолтні в'юшки-класи Django живуть в пакеті “django.views.generic”;
- 7ий: наш клас в'юшки унаслідується від класу UpdateView, адже ми реалізуємо форму редагування;
- 8ий: модель, з якою працюватиме наша сторінка - Student;
- 9ий: шаблон для сторінки редагування сторінка лежить за адресою “students/students_edit.html”;

- 11ий: метод, який повертає URL сторінку для редіректу після успішного збереження студента;
- 15ий: метод post ми також кастомізнули, щоб окремо обробляти кнопку “Скасувати” на формі; при її натиску ми переадресовуємо користувача на сторінку із списком студентів;
- 22ий: а всю важку роботу (валідація даних, збереження студента, підготовка помилок при некоректних даних) для нас зробить метод “post” батьківського класу.

Ось і все! Важко повірити, що 23 рядочки коду (разом із імпортами) - це все, що потрібно для редагування студента? Мені також було важко у це повірити, коли я вперше познайомився із даними формами.

А тепер скопіюємо шаблон для редагування студента із шаблону додавання студента:

Копіюємо шаблон додавання студента

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students/templates/students
2
3 $ cp students_add.html students_edit.html
```

Відкриваємо students_edit.html і приводимо до наступного вигляду:

Шаблон для роботи із в'юшкою-класом, students_edit.html

```
1 {% extends "students/base.html" %} 
2
3 {% load static from staticfiles %} 
4
5 {% block meta_title %}Редагувати Студента{% endblock meta_title %} 
6
7 {% block title %}Редагувати Студента{% endblock title %} 
8
9 {% block status_message %} 
10 {% if form.errors %}
```

```
11 <div class="alert alert-warning" role="alert">Будь-ласка, виправте н\  
12 аступні помилки</div>  
13 {% endif %}  
14 {% endblock %}  
15  
16 {% block content %}  
17  
18 <form action="{% url "students_edit" object.id %}" method="post" enc\  
19 type="multipart/form-data">  
20     {% csrf_token %}  
21  
22     {{ form.as_p }}  
23  
24     <input type="submit" value="Зберегти" name="add_button"  
25         class="btn btn-primary" />  
26     <button type="submit" name="cancel_button" value="Скасувати"  
27         class="btn btn-link">Скасувати</button>  
28 </form>  
29  
30  
31 {% endblock content %}
```

В даному шаблоні ми:

- оновили заголовки сторінки;
- кастомізували блок `status_message`, щоб виводити помилки валідації форми;
- вставили змінну форми “{{ form.as_p }}” аналогічно як зробили це з формою контакту адміністратора.

Нічого нового в даному шаблоні для нас не з'явилось. Зауважте, що даний клас в'юшки навіть не потребував класу форми, а згенерував для нас усі поля форми просто маючи атрибути “`model`” встановлений в значення `Student`. Атрибут класу `form_class` нам пригодиться, коли застосуємо Crispy Forms стилі до нашої форми.

Залишилась єдина зміна: оновити urls.py шаблон для сторінки із формою редагування:

urls.py

```
1 ...
2 from students.views.students import StudentUpdateView
3 ...
4 urlpatterns = patterns('',
5     ...
6     url(r'^students/(?P<pk>\d+)/edit/$',
7         StudentUpdateView.as_view(),
8         name='students_edit'),
9     ...
)
```

URL шаблони вміють працювати лише із об'єктами, що можна викликати (callable), тому передати напряму клас ми не можемо. Натомість, маємо в класах в'юшок статичний метод “as_view”, який спеціально призначений для повернення функції, що огортає наш клас у вигляд функції для URL шаблона.

Все. Отак, кількома рядками коду, навіть без декларації класу форми, а лише із класом в'юшки, ми маємо повноцінну форму редагування студента разом із валідацією та збереженням даних в базу.

На домашку: спробуйте зробити форму редагування визначивши окремо клас форми і вручну прописавши усі необхідні поля. Так само як ми це робили із формою контакту. Просто для практики і, щоб запам'ятати.

Crispy Forms стилі

На завершення даної секції приведемо нашу форму редагування студента до прийнятного вигляду. Знову ж таки використовуючи Crispy Forms аплікацію.

Усі дії аналогічні до тих, які ми проробляли з формою контакту адміна. Тому зробимо це доволі швидко і без додаткових дублюючих пояснень.

Якщо до цього моменту ми обходились без класу форми, то тепер маємо його створити, щоб встановити об'єкт класу FormHelper атрибутом у форму.

Ось як виглядатиме клас форми редагування студента:

Клас форми студента, views/students.py

```
31     self.helper.form_method = 'POST'
32     self.helper.form_class = 'form-horizontal'
33
34     # set form field properties
35     self.helper.help_text_inline = True
36     self.helper.html5_required = True
37     self.helper.label_class = 'col-sm-2 control-label'
38     self.helper.field_class = 'col-sm-10'
39
40     # add buttons
41     self.helper.layout[-1] = FormActions(
42         Submit('add_button', u'Зберегти', css_class="btn btn-primary"),
43         mary),
44         Submit('cancel_button', u'Скасувати', css_class="btn btn-link"),
45     )
46
47
48 class StudentUpdateView(UpdateView):
49     model = Student
50     template_name = 'students/students_edit.html'
51     form_class = StudentUpdateForm
52
53     def get_success_url(self):
54         return u'%s?status_message=Студента успішно збережено!' \
55             % reverse('home')
56
57     def post(self, request, *args, **kwargs):
58         if request.POST.get('cancel_button'):
59             return HttpResponseRedirect(
50             u'%s?status_message=Редагування студента відмінено!' \
51             %
52                 reverse('home'))
53
54         else:
55             return super(StudentUpdateView, self).post(request, *arg\
56 s, **kwargs)
```

Ми створили клас форми StudentUpdateForm, який унаслідується від ModelForm. Завдяки унаслідуванню наша форма отримує увесь функціонал по роботі із моделями і базою даних.

Подібним чином як ми це робили із формою контакту адміністратора, ми визначили об'єкт helper та “набили” його рядом атрибутів для кращого візуального представлення форми. Єдина відмінність полягає у тому, що тут ми додали дві кнопки та огорнули їх у лейаут FormActions (рядок 41). Він додасть спеціальний огортаючий тер div для тегів кнопок. Пізніше можна його використати для кастомних стилів.

Також в класі в'юшки, у рядку 51, ми під'єднали клас форми з допомогою атрибути form_class. Все решта у даному класі залишилось без змін.

Тепер черга за шаблоном. Що і як в ньому робити після застосування Crispy Forms ви уже знаєте:

Шаблон форми редагування студента із Crispy Forms

```
1  {% extends "students/base.html" %}

2
3  {% load static from staticfiles %}
4  {% load crispy_forms_tags %}

5
6  {% block meta_title %}Редагувати Студента{% endblock meta_title %}

7
8  {% block title %}Редагувати Студента{% endblock title %}

9
10  {% block status_message %}
11  {% if form.errors %}
12  <div class="alert alert-warning" role="alert">Будь-ласка, виправте н\
13  аступні помилки</div>
14  {% endif %}
15  {% endblock %}

16
17  {% block content %}
18
19  {% crispy form %}
```

20

21 {%- endblock content %}

Ми завантажили теги із Crispy Forms аплікації та замінили тег форми тегом “{%- crispy form %}”. Це все.

Знову оновіть сторінку із формою редагування студента і переконайтесь, що ваш результат є подібним на наступне зображення:

Редактувати Студента

Ім'я*	Новий
Прізвище*	Студент
По-батькові	побатькові
Дата народження*	2014-09-23
Фото	<input type="button" value="Choose File"/> No file chosen
Білет*	1232
Група*	Demo Group

Зберегти **Скасувати**

Форма редагування студента готова!

На домашнє завдання: перевести форму додавання студента також на форму моделей Django і користуватись єдиним шаблоном як для додавання, так і для редагування студента.

Ще одна вправа: спробувати переписати форму редагування студента без використання Django форм - повністю вручну. Так само, як ми це робили із формою додавання студента. З метою практики. Дасть вам гарне розуміння роботи із даними з бази та даними від користувача. Тут буде кілька нюансів,

яких не було із формою додавання. Плюс логіка по збереженні об'єкта в базу, звичайно, буде трохи іншою.

Як бачите, використання Django форм для роботи із моделями дозволило нам кількома рядками Python коду реалізувати доволі складну форму. В наступних главах книги ми ще матимемо справу із Django формами.

Видалення студента

Додавати та редагувати існуючого студента ми уже навчилися, і думаю, зможемо робити подібні речі доволі просто і швидко. На порядку денному - розібраться із видаленнями існуючих об'єктів з бази даних.

Для цього знову скористаємось максимально швидким способом: Django класами в'юшок для роботи із моделями - `DeleteView`²⁵⁴.

Наш власний клас буде унаслідуватись від `DeleteView` та декларуватиме два атрибути та один метод:

- `model`: клас моделі, над яким оперуватиме;
- `template_name`: шлях до шаблону, що міститиме підтверджуюче повідомлення про видалення студента;
- `get_success_url`: метод, що повертаєм шлях редіректу після успішного видалення студента.

Ось необхідні оновлення до нашого модуля із в'юшками `students.py`:

²⁵⁴<http://djbook.ru/rel1.7/ref/class-based-views/generic-editing.html#deleteview>

Клас-в'юшка для видалення студента

```
1 class StudentDeleteView(DeleteView):
2     model = Student
3     template_name = 'students/students_confirm_delete.html'
4
5     def get_success_url(self):
6         return u'%s?status_message=Студента успішно видалено!' \
7             % reverse('home')
```

Як бачите, нам знову не довелось писати жодної логіки по валідації форми чи видаленні об'єкта із бази даних.

Тепер скопіюємо один із існуючих кінцевих шаблонів у “students_confirm_delete.html”. Цей шаблон відображатиме повідомлення про підтвердження видалення студента. Важливо не одразу видаляти студента при кліку на Дію Видалити, а ще раз перепитати користувача про остаточне видалення. Адже ця дія остаточна:

Створюємо шаблон students_confirm_delete.html

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students/templates/students
2 $ cp students_edit.html students_confirm_delete.html
```

Відкриваємо новий шаблон і оновлюємо наступним кодом:

Кінцевий вигляд шаблону students_confirm_delete.html

```
1  {% extends "students/base.html" %}  
2  
3  {% load static from staticfiles %}  
4  
5  {% block meta_title %}Видалити Студента{% endblock meta_title %}  
6  
7  {% block title %}Видалити Студента{% endblock title %}  
8  
9  {% block content %}  
10  
11 <form action="{% url "students_delete" object.id %}" method="post">  
12   {% csrf_token %}  
13   <p>Ви дійсно хочете видалити студента "{{ object }}"?</p>  
14   <input type="submit" value="Так" name="delete_button"  
15     class="btn btn-danger" /><br /><br />  
16 </form>  
17  
18  {% endblock content %}
```

Як бачите код доволі простий. Ми додали форму, яка відсилає дані на себе ('% url "students_delete" object.id %') і містить запитання, чи дійсно користувач хоче видалити студента. Також додали кнопку “Так” для підтвердження дії.

Зверніть увагу, що в'юшка DeleteView також передає поточний об'єкт студента під змінною “object” в шаблоні. Ще одна цікава річ: коли вставляємо напряму об'єкт в шаблон (“{{ object }}”), тоді використовується його метод “__unicode__” для генерації стрічки-представлення об'єкта на нашій сторінці.

Залишилась остання річ: оновити URL шаблон для нашої сторінки підтвердження видалення студента:

DeleteView в urls.py

```
1 from students.views.students import StudentUpdateView, StudentDelete\\
2 View
3
4 ...
5 url(r'^students/(?P<pk>\d+)/delete/$',
6     StudentDeleteView.as_view(),
7     name='students_delete'),
8 ...
```

Що ми змінили в даному модулі:

- додали імпорт StudentDeleteView;
- змінили називу групи із “sid” студента на “pk”;
- змінили посилання на в’юшку: “StudentDeleteView.as_view()”.

Готово! У списку дій студента обираєте “Видалити” і видаліть одного із студентів. Переконтайтесь, що вигляд та функціонал форми працює коректно.

Видалити Студента

Ви дійсно хочете видалити студента "Vitaliy Podoba"?

Так

Видалення студента

На домашнє завдання: реалізувати аналогічним чином видалення груп.

Ще одне завдання: реалізувати видалення студента повністю вручну без Django форм.

Ну і для зовсім “просунутих”. Реалізувати можливість видалення одразу кількох студентів. Для цього треба буде додати чекбокси до списку із студентами, а також кнопки Видалити внизу або зверху даного списку.

Черговий раз переконуємось на скільки швидко та легко створювати стандартний функціонал по роботі із базою даних, якщо під руками Django форми. Звичайно, спочатку треба провести деякий час практикуючись із ними, але пізніше це окуповується з вторицею.

Кастомізація адміністративної частини Django

На завершення даної глави оглянемо, яким чином можна змінювати адмін інтерфейс Django. Спочатку розберемо теоретичні основи і можливості для кастомізації, а потім зробимо 2 практичні задачі:

- покращимо список студентів в адмінці функціями пошуку, фільтру та редагуванням;
- реалізуємо валідацію моделі студента, щоб не можна було призначити студента до групи, якщо він є старостою в іншій групі; тобто група має співпадати із тою, де даний студент є старостою.

Теорія адмінки

Django надає широкі можливості²⁵⁵ для кастомізації адміністративної частини.

²⁵⁵<http://djbook.ru/rel1.7/ref/contrib/admin/index.html>

Усі зміни відбуваються через власний Python клас, який повинен унаслідуватись від `ModelAdmin`.

Ось список найчастіше використовуваних атрибутів і методів даного класу для кастомізації адмін частини тієї чи іншої моделі:

- `actions`: список додаткових дій над елементами в списку моделей; зазвичай ці дії розробнику потрібно самому реалізувати;
- `actions_selection_counter`: показувати чи ні лічильник вибраних серед списку об'єктів;
- `exclude`: список полів, які виключити із форми редагування моделі;
- `fields`: список полів, які включити у форму редагування моделі;
- `form`: призначення кастомного класу форми; використовується для серйозніших змін у функціонал та вигляд форми редагування моделі; пізніше ми скористаємося даним атрибутом;
- `list_display`: список полів моделі для відображення у списку;
- `list_display_links`: які із полів моделі у списку будуть огорнуті в лінк, який вказуватиме на форму редагування об'єкта
- `list_editable`: список полів, які можна буде поредагувати прямо у списку об'єктів; не заходячи на форму редагування;
- `list_filter`: список полів для генерації фільтрів;
- `list_per_page`: кількість об'єктів на одній сторінці;
- `ordering`: список полів, по яких початково сортувати об'єкти;
- `readonly_fields`: список полів, які не можна буде редагувати на формі;
- `search_fields`: список полів, по яких можна буде робити пошук об'єктів;
- `view_on_site`: метод, що повертає URL адресу до фронт-енд в'юшки об'єкта.

Ще є багато інших атрибутів і методів, які, буквально, дозволяють повністю переписати форми і шаблони адмінки ваших моделей. Але їх розглядати тут не будемо.

Таким чином, достатньо визначити свій клас і прописати необхідні методи та атрибути всередині, щоб впливати на вигляд та функціонал адміністративної частини обраної моделі. А почнемо із покращення функцій для списку студентів.

Покращуємо список студентів в адмінці

Ось як виглядає список студентів на даний момент. Порівняємо його наприкінці даної секції, коли матимемо усі покращення:

Дія:	Уперед	0 з 12 обрано
<input type="checkbox"/> Студент		
<input type="checkbox"/> Андрій Прізвище1		
<input type="checkbox"/> Новий Студент		
<input type="checkbox"/> Новий Студент		
<input type="checkbox"/> Vitaliy Podoba		
...		

Дефолтна адмінка списку студентів

Ми уже раніше зареєстрували клас моделі студента для Django адмінки, і тепер залишається лише визначити окремий клас для адмін в'юшки і підв'язати її до моделі студента. Поки вона буде у нас порожня:

Адмін в'юшка, модуль `admin.py` в корені аплікації `students`

```

1 # -*- coding: utf-8 -*-
2 from django.contrib import admin
3
4 from .models import Student, Group
5
6 class StudentAdmin(admin.ModelAdmin):
7     pass
8
9 admin.site.register(Student, StudentAdmin)
10 admin.site.register(Group)

```

Ми створили клас `StudentAdmin` і унаслідували його від `ModelAdmin`, адже усі в'юшки в адмінці - це в'юшки для роботи із моделями. Крім того, ми передали даний клас другим аргументом у функцію “`register`”. Тепер модель студента зв'язана із новствореним класом в'юшки.

Ми готові до того, щоб “нафарширувати” наш клас корисними речима:

Покращуємо вигляд списку студентів

```
1 from django.core.urlresolvers import reverse
2
3 class StudentAdmin(admin.ModelAdmin):
4     list_display = ['last_name', 'first_name', 'ticket', 'student_gr\oup']
5     list_display_links = ['last_name', 'first_name']
6     list_editable = ['student_group']
7     ordering = ['last_name']
8     list_filter = ['student_group']
9     list_per_page = 10
10    search_fields = ['last_name', 'first_name', 'middle_name', 'tick\et',
11                      'notes']
12
13    def view_on_site(self, obj):
14        return reverse('students_edit', kwargs={'pk': obj.id})
```

Тепер давайте порядково ще раз пройдемось по доданих атрибутих і методах класу:

- 4ий рядок: відображаємо в заданому порядку Прізвище, Ім'я, Білет та Групу студента у списку;
- 5ий: поля Ім'я та Прізвище будуть залінковані на форму редагування студента;
- 7ий: поле Група можна тепер буде редагувати пряму на списку студентів; це пришвидшить дії адміністратора по розподілу студентів по групах;
- 8ий: по-замовчуванню список студентів буде посортований по прізвищу;
- 9ий: в правій колонці біля списку студентів з'явиться можливість фільтрувати студентів по групах; це зручно для роботи із студентами в контексті тої чи іншої групи;
- 10ий: вказуємо посторінковій навігації відображати 10 студентів на одній сторінці;

- 11ий: додаємо форму пошуку студентів, яка шукатиме серед усіх текстових полів у моделі студента: Прізвище, Ім'я, По-батькові, Білет та Додаткові нотатки;
- 15ий: додаємо кнопку “Переглянути на сайті” до форми редагування студента; таким чином адміністратор зможе одним кліком відвідати фронт-енд представлення поточного студента; у нашому випадку це форма редагування студента; даний метод “view_on_site” повертає URL адресу сторінки.

Все доволі просто і згідно тих визначень, які ми переглянули в попередній секції із теоретичним вступом.

Після усіх вищеперечислених змін ось як виглядатиме наш список студентів:

Прізвище	Ім'я	Білет	Група
Podoba	Vitaliy	234	demo 1 (Vitaliy Podoba)
Podoba	Vitaliy	234	Demo Group 1 (Новий Студент)
Podoba	Vitaliy	1234	demo 1 (Vitaliy Podoba)
student	demo	456	demo 1 (Vitaliy Podoba)

Список студентів покращений

Форма пошуку, фільтри, сортування, поля, редагування, посторінкова навігація - усе це та ще дещо, ми змогли зробити кількома стрічками коду у класі в'юшки управління виглядом студентів в адміністративній частині Django.

На домашнє завдання: кастомізуйте список груп в адміністративній частині Django подібним чином до того, як ми це зробили із списком студентів. Додайте функції пошуку та фільтру, щоб працювати із групами було значно зручніше.

Складніше завдання: розібратись із [діями²⁵⁶](#) в Django адмінці і реалізувати функцію копіювання обраних студентів у списку.

²⁵⁶<http://djbook.ru/rel1.7/ref/contrib/admin/actions.html>

Тепер перейдемо до дещо складнішого завдання.

Валідатор для поля Групи

Кожен із студентів повинен бути призначений в ту чи іншу групу. В той же час, деякі із студентів можуть бути обрані в якості старост для груп.

На даний момент на формі редагування студента в адміністративній частині ми можемо обирати будь-яку існуючу групу в полі Група. Насправді в реальному житті не буває так, щоб студент навчався в одній групі, а був старостою іншої групи. Проте, використовуючи наші адмін форми, ми можемо таке зробити. Потрібно виправити даний функціонал і узпечити адміністратора сайту від подібних помилок з налаштування студентів та груп.

Бути в курсі усіх студентів, груп та старост є непосильною задачею для людини, тому розв'язок даної проблеми повинен лежати на рівні коду, а не на рівні інструктажу персоналу нашої аплікації.

Це все можна виправити з допомогою додаткової валідації поля Група на формі редагування студента. При кожній спробі оновити студента ми будемо перевіряти чи обрана група співпадає із групою, в якій даний студента являється старостою. Звичайно, якщо даний студент не є старостою в жодній групі, тоді і перевіряти нічого не потрібно. Будь-яка група для нього є дозволеною.

Із специфікаціями даного завдання розібрались, тому переходимо безпосередньо до реалізації коду.

Щоб додати валідатор потрібно мати клас форми. На даний момент ми його ще не створили. Тому додаємо форму і пропишемо її в атрибут `form_class` нашої в'юшки, щоб зв'язати їх один із одним. Наводжу одразу кінцевий код валідаційного методу, а після цього розберемось в деталях:

Валідація поля student_group

```
1 # -*- coding: utf-8 -*-
2 from django.contrib import admin
3 from django.core.urlresolvers import reverse
4 from django.forms import ModelForm, ValidationError
5
6 from .models import Student, Group
7
8
9 class StudentFormAdmin(ModelForm):
10
11     def clean_student_group(self):
12         """Check if student is leader in any group.
13
14             If yes, then ensure it's the same as selected group."""
15         # get group where current student is a leader
16         groups = Group.objects.filter(leader=self.instance)
17         if len(groups) > 0 and \
18             self.cleaned_data['student_group'] != groups[0]:
19             raise ValidationError(u'Студент є старостою іншої групи.\n',
20             code='invalid')
21
22
23     return self.cleaned_data['student_group']
24
25 class StudentAdmin(admin.ModelAdmin):
26     list_display = ['last_name', 'first_name', 'ticket', 'student_group']
27     list_display_links = ['last_name', 'first_name']
28     list_editable = ['student_group']
29     ordering = ['last_name']
30     list_filter = ['student_group']
31     list_per_page = 10
32     search_fields = ['last_name', 'first_name', 'middle_name', 'ticket']
33
34     et,
```

```
35         'notes']
36     form = StudentFormAdmin
37
38     def view_on_site(self, obj):
39         return reverse('students_edit', kwargs={'pk': obj.id})
40
41 admin.site.register(Student, StudentAdmin)
42 admin.site.register(Group)
```

Давайте порядково пройдемось по останніх оновленнях:

- 9ий рядок: декларуємо клас StudentFormAdmin, який унаслідує від ModelForm;
- 11ий: у Django формах перед викликом методів `is_valid` та `is_invalid` викликаються методи валідації та перетворення даних із назвами `clean_`[назва поля]; ми скористались таким підходом та перекрили метод валідація саме поля групи “`clean_student_group`”; він не приймає жодних атрибутів, але поточний об'єкт форми має атрибут `cleaned_data`, яким ми і скористаємося, щоб дізнатись яку саме групу обрав користувач на формі редагування студента;
- 16ий: витягуємо усі групи з бази даних, в яких даний студента є призначений старостою;
- 17ий: якщо даний студент є старостою хоча б в одній групі, а також ця група не співпадає із групою, яка була щойно обрана на формі редагування...;
- 19ий: тоді викидаємо спеціальну помилку `ValidationError`, якій передаємо повідомлення для відображення користувачеві;
- 23ій: у випадку, якщо валідація проходить успішно, ми просто повертаємо значення обраного поля `student_group` із методу;
- 36ий: прив'язуємо наш клас форми до адмін в'юшки.

Тепер, щоб потестувати наш валідатор, потрібно відповідно налаштувати групу та студента. В одній із груп додайте студента в якості старости. Потім на формі редагування даного студента спробуйте обрати іншу групу і зберегти. При цьому ви повинні отримати повідомлення над полем: “Студент є старостою іншої групи.”.

Як розумієте валідація лише із сторони форми студента буде неповною, адже ще залишається можливість на формі групи вибирати старосту. Тому даю вам на самостійне опрацювання реалізацію валідатора поля Староста Групи. Даний валідатор не дозволить обрати студента, якщо він не належить до поточної групи (тобто його поле `student_group` не вказує на дану групу).

Ще одне завдання. Додайте аналогічну валідацію до наших власних фронтенд форм редагування студента та групи.

Не забуваємо про репозиторій коду і регулярні коміти окремими завданнями та із гарними пояснювальними коментарями.

Домашнє завдання

Протягом даної глави ми освоїли масу нового матеріалу. Давайте підсумуємо:

- робота із HTML формами;
- обробка форм, валідація, робота із помилками;
- використання Django форм та форм роботи із моделями;
- реалізація форм додавання, видалення та редагування об'єктів в базі;
- створення правильним статусних повідомлень та відповідей після дій над формами;
- використання Twitter Bootstrap стилів з допомогою Crispy Forms аплікації;

- кастомізація Django адміністративного інтерфейсу.

В кожній із секцій ви зустрічали далеко не одне домашнє завдання. Як ви розумієте, вони там не лише для прочитання, але й для самостійної реалізації. Тому виділіть собі кілька днів, а може і тижнів, щоб хоча б спробувати кожне із них.

А для тих, кому все дається занадто легко, ось ще декілька ідей і нагадувань:

- реалізувати усі (редагування, додавання та видалення) форми роботи із Студентами, Групами та Іспитами;
- реалізувати дані форми двома способами: повністю вручну та з допомогою Django форм роботи з моделями.

Я спеціально даю вам таку рутинну роботу. Адже з кількістю прийде якість і повне розуміння концепції. Зазвичай, якщо я щось не розумію, я його завчаю, і після цього розуміння неодмінно приходить.

Тому не лінуйтесь і результати вже будуть не за горами.

Також в коді, що йде із книгою можете знайти код форм менеджменту групами. Там почерпнете декілька нових фішок, які ми не оглянули в даній главі.

...

Таким чином, даною главою ми закінчуємо найбазовіший матеріал книги. Форми, моделі та в'юшки - це необхідний мінімум базових знань, без яких неможливо знайти роботи та й взагалі рухатись далі. Так, він не є достатнім, але без нього нікуди. Тому виділіть достатньо часу на ці три глави.

В наступній главі ми перейдемо до динамічних сторінок, реалізуємо закладку Відвідування, ознайомимось із мовою Javascript та технологією AJAX, зробимо випадаюче меню груп робочим.

9. Реалізуємо журнал відвідування: Javascript та AJAX в Django

До цієї глави ми розглянули три кити веб-розробки з Django: в'юшки, моделі та форми. Якщо весь цей матеріал ви добраче засвоїли, тоді освоїти решту глав у даній книзі вам буде значно простіше.

У даній главі ми з вами поговоримо про динамізацію веб-сторінок та розберемось із такими технологіями і мовами як:

- Javascript - мова програмування на стороні веб-браузера;
- jQuery - надбудова над Javascript для полегшення життя програмісту;
- AJAX - технологія реалізації запиту на сервер з допомогою мови Javascript;
- динамічний функціонал сторінок в контексті фреймворка Django.

Javascript зазвичай на українській мові транслітерують як Джаваскрипт. За переклад у варіанті Яваскрипт джавісти дуже критикують ;-)

Протягом глави реалізуємо наступний функціонал:

- закладку Відвідування із динамічним оновленням журналу;
- випадайку з групами;
- календар віджет для поля дати;
- форму додавання в режимі AJAX та в попап вікні, без перевантаження сторінки.

Кожне із даних практичних завдань вимагатиме від нас написання Javascript коду. Таким чином, вони підібрані спеціально для практики Javascript та AJAX функціоналу в нашійapplікації.

Вступ до Javascript, AJAX, jQuery

Почнемо, як завжди, із невеликої дози ввідної теорії. Для детальнішого вивчення теоретичного базису я буду наводити подальші посилання на корисні матеріали.

Мова Javascript

Буквально кількома параграфами тексту ми уже з вами ознайомились із даною мовою ще у другій главі даної книги. Але цього недостатньо і спробуємо в даній секції копнути трохи детальніше. Звісно, курсу по Javascript тут ви не знайдете, а лише найважливіші факти. Тому одразу рекомендую пройти один із безкоштовних онлайн курсів по мові Javascript, щоб приклади і код даної глави були вам більш зрозумілими.

Ось кілька посилань для детальнішого освоєння даної мови:

- відео: [Базовий курс для початківців²⁵⁷](https://www.youtube.com/watch?v=fXi49EkrMPI);
- [Онлайн посібник по Javascript²⁵⁸](http://learn.javascript.ru/);
- [ще один відео курс основ мови Javascript²⁵⁹](http://habrahabr.ru/company/hexlet/blog/205902/).

Відмінність з Python

Усе пізнається краще у порівнянні. Тому давайте порівняємо мову програмування Javascript із Python, адже базу мови Python ми уже маємо. Наводжу випадковим чином список фактів для порівняння:

- код Javascript запускається на стороні браузера, а Python - на сервері;

²⁵⁷ <https://www.youtube.com/watch?v=fXi49EkrMPI>

²⁵⁸ <http://learn.javascript.ru/>

²⁵⁹ <http://habrahabr.ru/company/hexlet/blog/205902/>

- змінні в Javascript оголошуються ключовим словом var; в Python їх оголошувати не потрібно взагалі;
- відступи в Javascript не є обов'язковими, а вкладені блоки коду огортаються у фігурні дужки;
- після виразу чи блоку вкладеного коду у мові Javascript потрібно ставити крапку з комою (,:);
- в Python є змінні булевівські True і False, в Javascript - true і false;
- в Javascript є лише один тип для чисел: дробові числа (floating point number);
- Javascript код може працювати по різному у різнихбраузерах в той час, як Python інтерпретатор кросплатформений;
- ключове слово, яке позначає порожнє значення у Python позначається як None; в Javascript маємо два позначення: undefined і null;
- в Javascript є лише тип списку; в Python також є незмінні списки - кортежі;
- те, що в Python декларується як тип даних словник, в Javascript є об'єктом;
- в мові Python функція декларується ключовим словом def; в Javascript - function;
- ключове слово в ООП мови Python - self в Javascript мові перетворюється у this;
- ООП в двох мовах кардинально різні; в Javascript є підхід так званих прототипів²⁶⁰;

В [даній статті²⁶¹](#) можете ознайомитись детальніше із відмінностями даних двох мов.

А загалом, обидві мови є інтерпретованими, динамічними та із власним збирачем сміття (garbage collector). Обидві володіють непоганим міксом імперативних, функціональних та ООП парадигм. То ж, я думаю, програмувати на Javascript, після освоєння мови Python, не складе особливих труднощів.

Суть мови Javascript

²⁶⁰<http://javascript.ru/tutorial/object/inheritance>

²⁶¹<https://blog.glyphobet.net/essay/2557>

Ми вже з вами знаємо, що код на Javascript зазвичай вставляється на сторінку з допомогою спеціальних тегів “script” в заголовку сторінки (тег “head”), або перед закриваючим тегом “body”. Знайшовши даний тег на сторінці, браузер піде на сервер за даною йому адресою в тезі, отримає файл, відкриє і спробує запустити код всередині нього.

Javascript код зазвичай виконує три функції:

- слідкує за змінами на сторінці та діями користувача;
- відправляє запити на сервер;
- оновлює елементи на веб-сторінці.

По-суті, мова Javascript використовується для побудови веб-аплікацій із багатим динамічним функціоналом та графічним інтерфейсом для користувача. Така аплікація стає дуже подібною по зручності до десктопної.

Події в Javascript

Щоб слідкувати за діями користувача в мові Javascript є концепція подій. Коли користувач клікає на елемент, скролить сторінку, рухає мишкою чи натискає клавішу на клавіатурі, Javascript код може відловлювати дані події і відповідним чином реагувати на них.

Найбільш поширеними подіями є:

- onchange: HTML елемент зазнав змін;
- onclick: користувач клацнув мишкою по елементу;
- onmouseover: курсор мишкої наведено над елементом;
- onmouseout: курсор мишкої забрано з елемента;
- onkeydown: користувач натиснув клавішу;
- onload: браузер закінчив завантажувати елемент; це може бути сторінка, фрейм, зовнішній файл чи зображення.

Крім того, можна реалізовувати свої власні типи подій.

Код, що моніторить події зазвичай можна реєструвати з допомогою одного із наступних варіантів:

- атрибутом на тег необхідного елемента;
- ззовні, через функцію attachEvent або addEventListener; назва методу залежить від браузера; так, Javascript код може відрізнятись для різних веб-переглядачів.

Приклад моніторингу події onclick

```
1 // через document ми доступаємось до API роботи із
2 // елементами на сторінці; в даному прикладі ми шукаємо
3 // елемент з id="my-link" і навішуємо йому обробник клика;
4 var mylink = document.getElementById('my-link');
5 mylink.addEventListener('click',
6     function(event) {
7         alert('my-link clicked!');
8     }
9 );
```

Проте ми не використовуватимемо дані методи, а користуватимемось високо-рівневими методами бібліотеки jQuery. Вона також дозволить нам мати одну версію коду для різних веб-браузерів.

Вплив на елементи сторінки

Javascript має ряд інструментів для того, щоб змінювати елементи на сторінці, а також керувати елементами самого браузера.

Коли браузер завантажує HTML код сторінки, він формує ієархічне дерево об'єктів, що відповідає HTML структурі документа. Даний набір об'єктів сторінки називається DOM (Document Object Model, Об'єктна Модель Документа). Якраз із даною моделлю наш Javascript і співпрацює, щоб мати вплив на вигляд сторінки.

На сторінці наш Javascript код має доступ до глобальної змінної document, яка і надає доступ до решти елементів, атрибутів та властивостей сторінки. Через них ми можемо шукати елементи на сторінці, видаляти, додавати нові та змінювати існуючі.

В наступному прикладі ми знаходимо елемент на сторінці із 'id="apple"' і ховаємо його:

Приклад зміни елемента на сторінці

```
1 var apple = document.getElementById('apple');  
2 apple.style.display = 'none';
```

Крім DOM, Javascript також має доступ до різноманітних аспектів самого браузера. Ось деякі із найпоширеніших речей, до яких ми маємо доступ із Javascript коду:

- screen: робочий екран браузера;
- location: об'єкт, що відповідає за адресу поточної сторінки;
- history: історія навігації користувача в інтернеті;
- window: вікно браузера.

Усі ці об'єкти називаються BOM (Browser Object Model, Об'єктна Модель Браузера). Таким чином через мову Javascript ми маємо усе необхідне, щоб реалізувати по-справжньому динамічні інтерфейси у веб-переглядачі.

Технологія AJAX

Сама мова Javascript сприймалась досить несерйозно до моменту, коли Джесі Гаррет у 2005 році опублікував статтю “Новий підхід до веб-застосунків”. У ній він описав підхід [AJAX²⁶²](#), коли можна було комунікувати із сервером без повного перевантаження сторінки.

Дослідно AJAX розшифровується як Асинхронний Javascript та XML. Javascript - керує логікою і робить запит на сервер, а XML - це формат передач даних.Хоча XML вже давно не є єдиним форматом передачі даних. Окрім нього використовують JSON, HTML, Javascript код та простий текст.

²⁶²<http://uk.wikipedia.org/wiki/AJAX>

Хоч я і називатиму часом **AJAX** - технологією, проте він є лише підходом. Він дозволяє відправляти запити на сервер без перевантаження всієї сторінки. Це дозволяє будувати динамічні інтерфейси користувача у браузері.

На самих початках використовували приховані фрейми (теги frame та iframe), щоб здійснювати запити на сервер у фоновому режимі. Проте, згодом, компанія Microsoft додала спеціальний об'єкт [ActiveX²⁶³](#) до свого браузера Internet Explorer 5.0, який дозволяв тепер робити запити значно простіше та контролюваніше. Слідом за ним в компанії Mozilla інтегрували подібну можливість для запитів через об'єкт [XMLHttpRequest²⁶⁴](#).

З допомогою даних об'єктів ми, з нашого Javascript коду, можемо кількома стрічками робити запити на сервер різної складності і різного формату, отримувати відповідь і ефективно відповідати на дії користувача на веб-сторінці.

При цьому одним із найпопулярніших форматів передачі даних між клієнтом та сервером став формат [JSON²⁶⁵](#). Якщо по-простому, то в термінах мови програмування Python, JSON - це словник або список із даними. Дані можуть бути вкладеними, а ключ словника може містити ще один словник. Ми використовуватимемо JSON, а також формат HTML для передачі даних між клієнтом та сервером при AJAX запитах.

Послідовність дій в AJAX підході

Зробити гарну динамічнуapplікацію з інтуїтивним інтерфейсом, зрозумілими діями та відповідями користувачу не так вже й просто. Тому, для початку, варто притримуватись базових принципів побудови користувацького інтерфейсу при реалізації AJAX функціоналу.

Уявимо, що користувач вводить дані на формі і постить її без перевантаження всієї сторінки, тобто з допомогою AJAX підходу. Ось, що має Javascript код з хорошим AJAX підходом:

- динамічно валідувати кожне поле форми ще до відправки на сервер;
- якщо дані некоректні, не постити форму, а вказати на помилки;
- якщо дані коректні, зробити запит на сервер із заповненими даними;

²⁶³<http://uk.wikipedia.org/wiki/ActiveX>

²⁶⁴<http://uk.wikipedia.org/wiki/XMLHttpRequest>

²⁶⁵<http://uk.wikipedia.org/wiki/JSON>

- в той час як іде запит потрібно заблокувати кнопки і поля на формі, щоб уникнути подальших змін;
- також показати індикатор, що запит в процесі;
- коли прийшли дані від сервера - обробити їх і, якщо пост форми був успішно оброблений на сервері, показати повідомлення з результатом;
- також очистити форму;
- якщо не успішно - вказати на помилки і не очищати поля форми;
- в будь-якому випадку потрібно також заховати індикатор того, що запит в процесі, адже він уже закінчився;
- також знову активувати кнопки і поля форми.

Досить довгий список. Правда? Так. Побудова по-справжньому простих у використанні форм - це непроста задача. Проте є цілі фреймворки, які вже це все роблять для нас. Потрібно лише їх інтегрувати.

Міжбраузерні війни

Ще з перших версій веб-переглядачів компанії Microsoft та Mozilla постійно конкурували між собою та реалізовували різноманітні інтегровані в браузер веб-протоколи та стандарти по-різному. Через ці війни постійно страждали (і дотепер страждають) веб-розробники.

Як результат даної конкуренції досить часто ваш Javascript, написаний і протестований в одному веб-переглядачі, може ламатись у іншому. Щоб уникати даних проблем програмісти, зазвичай, використовують бібліотеки-надбудови над мовою Javascript, які абстрагують їх від різниці між браузерами.

Ми використовуватимемо одну із найпопулярніших Javascript бібліотек **jQuery²⁶⁶**. Окрім того, ми уже підключили і використовуємо Twitter Bootstrap фреймворк, який також надає цілий ряд “заряджених” Javascript-ом динамічних віджетів.

²⁶⁶<https://ru.wikipedia.org/wiki/JQuery>

Бібліотека jQuery

Бібліотека [jQuery²⁶⁷](#) захистить нас від відмінностей між різними веб-переглядачами. Але це далеко не один плюс даної бібліотеки. Разом з нею ми розроблятимемо Javascript функціонал в рази швидше з допомогою полегшеної роботи з:

- елементами на сторінці;
- AJAX підходом;
- фреймворком подій;
- додатковими плагінами;
- Twitter Bootstrap віджетами;
- візуальними ефектами на сторінці.

Весь Javascript код, який ми писатимемо в даній главі буде написаний з використанням функцій jQuery бібліотеки.

Типовим прикладом використання даної бібліотеки є ланцюжкова побудова дій (викликів методів) над знайденими на сторінці елементами:

Типове використання jQuery

```
1 // знак долара, це функція пошуку елемента на сторінці
2 // за даним селектором
3 $("div.test").add("p.quote").addClass("blue").slideDown("slow");
```

У вищезаведеному прикладі ми знайшли і отримали тег div із класом “test” на сторінці. Додали всередину нього тег параграфа p із класом “quote”. Потім додали ще один клас до цього параграфа - “blue”. І вкінці показали даний елемент на сторінці з допомогою ефекту розгортання елемента плавно зверху донизу.

...

²⁶⁷<https://ru.wikipedia.org/wiki/JQuery>

Скрипт jQuery бібліотеки уже є у нас на сторінці. Для віджета календаря ми заінсталюємо додатково плагін [Bootstrap Datepicker²⁶⁸](#), а для динамізації форми скористаємось jQuery плагіном [jQuery Form Plugin²⁶⁹](#). Для реалізації випадайки груп нам потрібно буде встановлювати та змінювати куки в браузері. З допомогою плагіна [jQuery Cookie²⁷⁰](#) ми з легкістю зможемо маніпулювати потрібними куками.

Інструменти розробки Javascript

Для написання Javascript коду можете продовжувати використовувати той же ж редактор, який використовували для мови Python. Швидше за все він також має підсвітку і для Javascript.

Тестувати розробку будемо лише в браузері Firefox, а дебажити з допомогою класного інструмента Firebug. З ним ми уже з вами познайомились впродовж попередніх глав. У цій главі ми особливо інтенсивно користуватимемось закладками Console та Script.

З тим, як дебажити Javascript код можете більше ознайомитись у [даній статті²⁷¹](#).

AJAX в контексті Django

Існує маса аплікацій та бібліотек, щоб полегшити розробку AJAX інтерфейсів в Django:

- [Django Ajax²⁷²](#);
- [DAjaxProject²⁷³](#);
- [Simple Django Ajax Endpoints²⁷⁴](#).

²⁶⁸<http://eonasdan.github.io/bootstrap-datetimepicker/>

²⁶⁹<http://malsup.com/jquery/form/>

²⁷⁰<http://plugins.jquery.com/cookie/>

²⁷¹http://www.w3schools.com/js/js_debugging.asp

²⁷²<https://github.com/yiceruto/django-ajax>

²⁷³<http://www.dajaxproject.com/>

²⁷⁴<https://github.com/joestump/django-ajax>

Тут²⁷⁵ можете ознайомитись із повним списком існуючих наробок в напрямку AJAX функціоналу для веб-фреймворка Django.

Проте ми не будемо використовувати додаткові бібліотеки чи аплікації. Нам потрібно отримати базове розуміння Javascript та технології AJAX, тому по-максимуму використовуватимемо власний код та те, що у нас буде доступне на сервері з Django. А клієнтську сторону, звичайно, доповнимо усіма бібліотека згаданими у попередніх секціях.

Django уже дає нам кілька спрощень при роботі із AJAX:

- “request.is_ajax()”: метод на об’єкті запиту, який повертає True для запитів типу AJAX; даний метод працює на базі спеціального заголовка, який браузер відсилає на сервер у випадку запиту через Javascript код;
- “json”: Python бібліотека, з допомогою якої будемо перетворювали Python дані у JSON формат перед відправкою на клієнт;
- “django.core.serializers.json”: перетворювач даних з бази в JSON формат.
- “django.http.JsonResponse”: об’ект відповіді, що правильно і з правильними заголовка надішле на клієнт JSON дані.

Таким чином, уже маємо в Django усе необхідне для того, щоб:

- відрізняти звичайні запити від AJAX запитів;
- правильно форматувати AJAX відповіді;
- розбирати дані на сервері під час AJAX запиту.

Тепер ми готові до практики. В процесі розберемо усі нюанси детальніше. А починаємо із розробки закладки Відвідування.

Перед переходом до практики хочу ще раз нагадати. Якщо приклади Javascript коду будуть в практичній роботі не зрозумілі, повертайтесь до теоретичних матеріалів по даній мові. Або при кожній новій функції чи парадигмі даної мови, яку братимемо під час того чи іншого прикладу, звертайтесь до

²⁷⁵<https://code.djangoproject.com/wiki/AJAX>

онлайн матеріалів, щоб підтягнути теоретичну базу з необхідної теми.

Закладка Відвідування

В даній секції реалізуємо закладку нашої аплікації під назвою Відвідування. На ній викладач зможе вести облік відвідування студентів. В реальному житті такий журнал ведеться для кожного окремого предмета окремим викладачем, але у нашому випадку ми зробимо трохи простіше і будемо вести облік відвідування закладу загалом:

The screenshot shows a web application titled "Сервіс Обліку Студентів". At the top, there are three tabs: "Студенти" (selected), "Відвідування" (highlighted in blue), and "Групи". A dropdown menu "Група:" is set to "Усі Студенти". Below the tabs, the title "Облік Відвідування" is displayed, followed by a date range "Жовтень 2014". The main area is a grid representing student attendance from October 1st to 31st. The columns are labeled with days of the week (Ср, Чт, Пт, Сб, Нд) and dates (1 through 31). The rows are labeled with student names: "Подоба Віталій", "Корост Андрій", and "Притула Тарас". Each cell contains a checkbox indicating attendance status. The grid shows that student 1 attended on most days, while students 2 and 3 had more absences.

#	Студент	Ср 1	Чт 2	Пт 3	Сб 4	Нд 5	Пн 6	Вт 7	Ср 8	Чт 9	Пт 10	Сб 11	Нд 12	Пн 13	Вт 14	Ср 15	Чт 16	Пт 17	Сб 18	Нд 19	Пн 20	Вт 21	Ср 22	Чт 23	Пт 24	Сб 25	Нд 26	Пн 27	Вт 28	Ср 29	Чт 30	Пт 31
1	Подоба Віталій	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>							
2	Корост Андрій	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
3	Притула Тарас	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>																					

© 2014 Сервіс Обліку Студентів

Закладка Відвідування

Щоб реалізувати дану закладку потрібно:

- створити відповідну модель, яка буде зберігати дані про відвідування;
- набити теги для шаблону закладки;
- реалізувати AJAX запис відвідування при кліку по чекбоксах;
- реалізувати усю логіку з відображенням студентів у списку включно із навігацією по місяцях та посторінковою навігацією студентів;
- ну і звичайно додати нову в'юшку до URL шаблонів.

Модель для журналу

Варіантів як реалізувати структуру даних для журналу є багато. Наприклад, можна створити один об'єкт, який зберігатиме усі записи про відвідування усіх студентів за уесь рік. А можна мати одну модель для одного студента. Або одну модель для одного студента та на один місяць. Або, якщо зовсім роздрібнити, один об'єкт (рядок в таблиці) для кожного окремого дня відвідування і окремого студента.

Давайте зупинимось на серединному варіанті, який дозволить нам не спамити таблицю в базі масою рядків і в той же час не реалізовувати модель із сотнями полів чи з полями складної структури (як от списки). Наша модель буде називатись MonthJournal і міститиме дані про відвідування лише одного студента протягом місяця.

В даній моделі ми матимемо список булеанівських полів на кожен день місяця (так, 31 поле), референс на студента, а також поле дати, яке прив'язуватиме поточний об'єкт журналу до того чи іншого місяця. Якщо пам'ятаєте, журнал відвідування в школі був поділений на сторінки і кожна сторінка містила список усіх студентів та облік відвідування рівно за місяць. Так само ми реалізуємо інтерфейс закладки форми. Таким чином, наша модель Місячного Журналу ідеально підходить під наш журнал.

Подумайте самостійно над тим, як саме ви б зберігали журнал відвідування у базі даних. Можливо існує більш зручний формати, ніж ми обрали в даній секції.

З'ясувавши структуру даних, необхідну для закладки Відвідування, переходимо до діла. Створимо модуль для моделі журналу:

Модуль для моделі журналу

```
1 # заходимо в пакет із моделями
2 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students/models
3 # створюємо файл для модуля моделі журналу
4 $ touch monthjournal.py
```

Відкриваємо новостворений файл в редакторі і набиваємо код для моделі нашого журналу:

Модель MonthJournal

```
1 # -*- coding: utf-8 -*-
2 from django.db import models
3
4
5 class MonthJournal(models.Model):
6     """Student Monthly Journal"""
7
8     class Meta:
9         verbose_name = u'Місячний Журнал'
10        verbose_name_plural = u'Місячні Журнали'
11
12        student = models.ForeignKey('Student',
13            verbose_name=u'Студент',
14            blank=False,
15            unique_for_month='date')
16
17        # we only need year and month, so always set day to first day of\
18        the month
19        date = models.DateField(
20            verbose_name=u'Дата',
21            blank=False)
22
23        # list of days, each says whether student was presene or not
24        present_day1 = models.BooleanField(default=False)
```

```
25     present_day2 = models.BooleanField(default=False)
26     present_day3 = models.BooleanField(default=False)
27     present_day4 = models.BooleanField(default=False)
28     present_day5 = models.BooleanField(default=False)
29     present_day6 = models.BooleanField(default=False)
30     present_day7 = models.BooleanField(default=False)
31     present_day8 = models.BooleanField(default=False)
32     present_day9 = models.BooleanField(default=False)
33     present_day10 = models.BooleanField(default=False)
34     present_day11 = models.BooleanField(default=False)
35     present_day12 = models.BooleanField(default=False)
36     present_day13 = models.BooleanField(default=False)
37     present_day14 = models.BooleanField(default=False)
38     present_day15 = models.BooleanField(default=False)
39     present_day16 = models.BooleanField(default=False)
40     present_day17 = models.BooleanField(default=False)
41     present_day18 = models.BooleanField(default=False)
42     present_day19 = models.BooleanField(default=False)
43     present_day20 = models.BooleanField(default=False)
44     present_day21 = models.BooleanField(default=False)
45     present_day22 = models.BooleanField(default=False)
46     present_day23 = models.BooleanField(default=False)
47     present_day24 = models.BooleanField(default=False)
48     present_day25 = models.BooleanField(default=False)
49     present_day26 = models.BooleanField(default=False)
50     present_day27 = models.BooleanField(default=False)
51     present_day28 = models.BooleanField(default=False)
52     present_day29 = models.BooleanField(default=False)
53     present_day30 = models.BooleanField(default=False)
54     present_day31 = models.BooleanField(default=False)
55
56     def __unicode__(self):
57         return u'%s: %d, %d' % (self.student.last_name, self.date.mo\
58 nth,
59                         self.date.year)
```

Давайте пройдемось по найбільш цікавих моментах даного модуля:

- 5ий рядок: декларуємо клас моделі журналу під назвою MonthJournal; як завжди унаслідуємось від базового класу Model;
- 8ий: прописуємо у вкладеному мета класі назви нашої моделі для адмінки Django;
- 12ий: поле-прив'язка до об'єкта студента; кожен студент матиме свої об'єкти журналу;
- 15ий: ми скористалися новим для нас атрибутом поля `unique-for-month`²⁷⁶; він вказує, що дане поле студента повинне бути унікальним в парі із полем date; але, при цьому, унікальним у відношенні до місяця даної дати; це означає, що не може бути в таблиці двох рядків журналу з однимаковим студентом і однаковим місяцем; завдяки даному полю ми на рівні бази даних та Django форм вказали не пропускати дублюючі журнали для студента за однакові місяці;
- 19ий: поле дати, яке вказуємо на місяць даного об'єкта журналу; нас цікавлять лише рік та місяць даного поля, а день завжди з коду встановлюватимемо у перший день місяця, адже об'єкт журналу створюється новий щомісяця;
- 24ий: починаємо список булеванівських полів, де кожне поле відповідає за один день у місяці; усього 31 поле; якщо студент був присутній у даний день, тоді дане поле, що відповідає цьому дню, буде встановлене у True;
- 56ий: метод `__unicode__` для того, щоб гарно відобразити об'єкт журналу в адмінці; ми показуємо прізвище студента, місяць та рік журналу.

На домашнє завдання: в циклі автоматизувати додавання усіх булеванівських полів. Це завдання на знання мови Python та те, як можна працювати із класами. Підказка: потрібно встановлювати атрибути класу поза ним. Клас в мові Python також є об'єктом.

²⁷⁶<https://docs.djangoproject.com/en/1.7/ref/models/fields/#unique-for-month>

Не відходячи від каси, зареєструємо нову модель для адміністративної частини Django:

Додаємо реєстрацію моделі MonthJournal в admin.py

```
1 # -*- coding: utf-8 -*-
2
3
4 from django.contrib import admin
5
6 ...
7
8 from .models import Student, Group, MonthJournal
9
10 ...
11
12 admin.site.register(MonthJournal)
```

І ще маємо додати імпорт MonthJournal класу в модулі __init__.py на рівні пакету моделей models:

models/__init__.py

```
1 from .monthjournal import MonthJournal
```

Переконайтесь, що ваше віртуальне середовище активоване і запускайте тепер міграційні скрипти:

Синхронізуємо модель MonthJournal в базу даних

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb
2 $ python manage.py makemigrations
3 $ python manage.py migrate
```

Запускайте ваш Django сервер і зайдіть в адмінку аплікації. Переконайтесь, що серед доступних моделей аплікації Students з'явився новий запис “Місячні Журнали”:

Адміністрування сайту

Students
Групи
Місячні Журнали
Студенти

Модель MonthJournal Готова!

Таким чином, ми закінчили із реалізацією моделі журналу і можемо переходити до підготовки шаблону для закладки відвідування.

Робимо заготовку

Додамо порожню в'юшку із під'єднаним шаблоном і підключимо її до URL адреси “/journal”.

Як завжди, копіюємо новий шаблон із існуючого, щоб зекономити на набивці спільніх блоків:

Копіємо шаблон для закладки Відвідування

```

1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students/templates/students
2
3 $ cp students_confirm_delete.html journal.html

```

Відкриваємо файл journal.html і готуємо базу:

Заготовка для шаблону відвідування, journal.html

```
1  {% extends "students/base.html" %}  
2  
3  {% load static from staticfiles %}  
4  
5  {% block meta_title %}Облік Відвідування{% endblock meta_title %}  
6  
7  {% block title %}Облік Відвідування{% endblock title %}  
8  
9  {% block content %}  
10  
11 Скоро тут буде журнал відвідування студентів.  
12  
13 {% endblock content %}
```

Все доволі просто і так, як ми це робили для попередніх шаблонів.

Підключимо наш шаблон до в'юшки. Для цього спочатку створимо новий модуль для в'юшок журналу:

Створюємо модуль для в'юшки журналу

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students/views  
2 $ touch journal.py
```

Відкриваємо його у редакторі та наповнюємо наступним кодом:

Заготовка для в'юшки журналу, journal.py

```
1 from django.views.generic.base import TemplateView  
2  
3 class JournalView(TemplateView):  
4     template_name = 'students/journal.html'
```

В даному випадку ми скористаємось в'юшкою у вигляді класу, а не функції. Для цього унаслідувались від базового класу TemplateView і прописали в класі атрибут template_name. Він вказує на шлях до шаблона. Поки все.

На завершення потрібно додати URL шаблон для нової сторінки:

URL шаблон для закладки Відвідування, urls.py

```
1 ...
2
3 from students.views.journal import JournalView
4
5 ...
6
7 url(r'^journal/$', JournalView.as_view(), name='journal'),
8 ...
9 ...
```

Тепер у вашому браузері спробуйте зайти на закладку Відвідування. Результат повинен бути подібним на оце:

Сервіс Обліку Студентів

Група: Усі Студенти

Студенти Відвідування Групи Контакт

Облік Відвідування

Скоро тут буде журнал відвідування студентів.

© 2014 Сервіс Обліку Студентів

Заготовка закладки Відвідування

Розробляємо шаблон

То ж, які елементи нам потрібно закласти в наш шаблон відвідування:

- основний елемент - це список студентів і відповідний список днів, щоб відмічати денну присутність;

- навігація по місяцях;
- навігація посторінкова по студентах;
- а також, якщо студентів немає ще в базі, тоді показуватимемо дефолтне повідомлення, яке запросять викладача додати першого студента в базу.

По-замовчуванню наш журнал відвідування показуватиме поточний місяць. Проте, якщо перейти на дану сторінку із спеціальним параметром в URL адресі, тоді можна отримувати інші місяці. Це потрібно для того, щоб викладач міг переглядати історію відвідування студентів протягом інших місяців.

Ось кінцевий код шаблону із вищеперечисленими елементами:

Шаблон для журналу відвідування

```
1  {% extends "students/base.html" %}  
2  
3  {% load static from staticfiles %}  
4  
5  {% block meta_title %}Облік Відвідування{% endblock meta_title %}  
6  
7  {% block title %}Облік Відвідування{% endblock title %}  
8  
9  {% block content %}  
10  
11  {% if not students %}  
12  
13  <div class="alert alert-warning">  
14      Поки немає жодного студента в базі. Будь-ласка,  
15      <a href="{% url "students_add" %}">додайте</a> першого.  
16  </div>  
17  
18  {% else %}  
19  
20  <p id="journal-nav" class="text-center">  
21    <a href="?month={{ prev_month }}"  
22      title="Попередній Місяць">&larr;</a>
```

```
23 <!-- TODO: translate month name -->
24 <strong>{{ month_verbose }} {{ year }}</strong>
25 <a href="?month={{ next_month }}"
26     title="Наступний Місяць">&rarr;;</a>
27 </p>
28
29 <table class="table table-hover table-striped table-bordered"
30     id="students-journal">
31
32     <thead>
33         <tr class="header">
34             <th class="num">#</th>
35             <th class="name">Студент</th>
36             {% for day in month_header %}
37                 <th class="day-header">
38                     <!-- TODO: translate day name -->
39                     {{ day.verbose }}
40                     <br />
41                     {{ day.day }}
42                 </th>
43             {% endfor %}
44         </tr>
45     </thead>
46
47     <tbody>
48
49         {% csrf_token %}
50
51         {% for student in students %}
52             <tr>
53
54                 <td>
55                     {% if is_paginated %}
56                         {{ page_obj.start_index|add:forloop.counter|add:"-1" }}
57                     {% else %}
```

```
58          {{ forloop.counter }}  
59      {% endif %}  
60  </td>  
61  
62  <td>  
63      <a title="Редагувати" target="_blank"  
64          href="{% url "students_edit" student.id %}">  
65          {{ student.fullname }}  
66      </a>  
67  </td>  
68  
69      {% for day in student.days %}  
70  <td class="day-box">  
71      <input type="checkbox" data-student-id="{{ student.id }}"  
72          data-date="{{ day.date }}" data-url="{{ student.updat\\  
73 e_url }}"  
74          value="1"{{ if day.present }} checked="checked" {{ en\\  
75 dif %}} />  
76  </td>  
77  {% endfor %}  
78  
79  </tr>  
80  {% endfor %}  
81  
82  </tbody>  
83  </table>  
84  
85  {% endif %}  
86  
87  {% with object_list=students %}  
88      {% include "students/pagination.html" %}  
89  {% endwith %}  
90  
91  {% endblock content %}
```

Давайте пройдемось в деталях по ньому, адже він містить багато нових і непростих елементів:

- 9ий рядок: усе новеньке, що ми додали, прийшло в блок content, тому починаємо із нього;
- 11ий: важливо не лише показувати список студентів, але й зручним для користувача чином відображати стани, коли ще немає даних в базі; тому, якщо поки немає студентів, ми показуємо користувачу повідомлення і запрошуємо додати першого студента; щоб упростити життя користувачу, ми одразу даємо лінк на форму додавання студента;
- 18ий: якщо ж список із студентами не є порожнім, тоді відображаємо наш журнал відвідування;
- 20ий: а починаємо із навігаційного елемента, який дозволить викладачу переглядати журнал відвідування; в огортаючий тег “r” ми додали “id”, щоб пізніше, при потребі, можна було прив’язатись стилями або Javascript кодом; також додали Twitter Bootstrap клас “text-center”, який вирівняє нам навігаційні елементи по центру сторінки; даний елемент містить два лінки на попередній і наступний місяці, а також назву поточного місяця з роком;
- 21ий: лінк на попередній місяць; це лінк на ту саму сторінку із журналом, але з додатковим URL параметром “month”; він вказуватиме на місяць, який потрібно відобразити на сторінці; тут ми використовуємо змінну, яку пізніше підготуємо у в’юшці - правильно відформатована дата першого дня попереднього місяця;
- 24ий: вставляємо назву місяця та рік, які в даний момент відображаємо в журналі відвідування; ці змінні нам також треба буде підготувати у в’юшці; зауважте TODO нотатку: в даний момент ми матимемо назву місяця лише англійською, а в наступних главах ми навчимось правильно перекладати дані слова;
- 25ий: аналогічний лінк, але цього разу на наступний місяць;
- 29ий: починаємо основну частину сторінки - таблицю із самим журналом; таблиця матиме рядки із студентами, для кожного з яких показуватимемо набір чекбоксів для кожного дня місяця; до таблиці ми додали кілька заготовлених у Twitter Bootstrap класів: “table” - дефолтні стилі від фреймворка для таблиці, “table-hover” - підсвічуємо рядки таблиці

при наведені на них мишкою, “table-striped” - чергуємо фони рядків таблиці у вигляді зебри (це полегшує візуальне сприйняття таблиці), “table-bordered” - додаємо до таблиці межі; також до тегу таблиці додали атрибут “id”, щоб пізніше робити AJAX запити із Javascript і, з легкістю, шукати дані для запитів;

- 32ий: `thead` - починаємо заголовок таблиці, який містить порядковий номер студента, ім’я та список днів у місяці;
- 36ий: пробігаємось по списку днів у місяці і готуємо шапку таблиці із тегів “`th`”; список днів ми приготуємо у ’юшці; кожен елемент списку буде словником із ключами `verbose` - абревіатура дня в тижні, `day` - день місяця; показуватимемо дані елементи на різних рядках в одній комірці;
- 47ий: `tbody` - починаємо саме тіло таблиці; вставляємо елемент CSRF захисту; ним скористаємось при формуванні AJAX запиту; так, самого тегу форми нам не потрібно, запит робитимемо самостійно із Javascript коду, без форми;
- 51ий: пробігаємось по переданих в шаблоні студентах (вони будуть уже погруповані для посторінкової навігації) та для кожного із них вставляємо комірку із номером студента, ім’ям та даними про відвідування кожного дня місяця;
- 55ий: якщо є більше, ніж одна сторінка студентів, тоді нам це треба врахувати при підготовці порядкового числа студента у списку; змінні `is_paginated` та `page_obj` нам передасть код, що формуватиме пагінацію студентів; змінна `forloop` передається всередину тіла циклу Django шаблоном; її атрибут `counter0` вказує на порядковий номер ітерації починаючи з нуля; таким чином, наприклад, будучи на другій сторінці із студентами, ми почнатимемо облік студентів не з 1, а з 11 (адже показуватимемо 10 студентів на сторінці);
- 58ий: якщо ж студентів недостатньо для посторінкової навігації, тоді просто відображаємо порядковий номер студента користуючись атрибутом `counter`, який починає відлік з 1;
- 63ій: вставляємо повне ім’я студента; огортаємо його в лінк, який вказуватиме на форму редагування студента; кожен студент матиме ключі “`fullname`” та “`id`”, якими ми тут успішно скористались;
- 69ий: це мабуть найцікавіше місце у нашему шаблоні журналу; кожен студент матиме ключ під назвою “`days`”, який міститиме список днів

місяця та те, чи він був присутнім у цей день; ми пробігаємо по усіх днях і для кожного дня формуємо комірку в рядку;

- 70ий: до тегу комірки додаємо клас “day-box”, щоб пізніше застосувати стилі і зробити вигляд таблиці кращим;
- 71ий: з допомогою чекбоксів ми дозволимо користувачу відмічати ті дні, коли студент був присутній; пізніше, на кожен такий клік по даному елементу ми прив’яжемо Javascript код, який буде записувати дані кліків в базу даних (а саме в MonthJournal таблицю); для того, щоб зробити даний запис потрібно знати, якого студента стосується дана зміна, який день мається на увазі та був студент присутнім чи відсутнім; атрибути, що починаються з префікса “data-“, якраз призначенні для того, щоб встановлювати додаткові дані для подальшої роботи із Javascript кодом; при кожному кліку ми вичитуватимемо ці дані і відсилатимемо на сервер: “data-student-id” - id студента, “data-date” - дата, “data-url” - адреса на сервері, куди робити запит;
- 74ий: значення поля встановлюємо в одиничку, а маркуємо його як “checked”, якщо студент був присутнім в цей день; усі дані витягуємо із студента та його даних всередині “days”; іх заготуємо пізніше на сервері у нашій в’юшці;
- 87ий: тут ми винесли код щодо посторінкової навігації в окремий файл; а також передаємо список студентів під назвою object_list; щоб зробити pagination.html універсальним і робочим для усіх наших сторінок, де ми потребуємо такої навігації, він повинен отримувати список елементів під однією і тією ж змінною; тег “with” нам класно у цьому допоміг; про навігацію і її “рефакторинг” (з англ. “refactor” - реорганізація, зміна, оновлення) ми поговоримо пізніше; якщо ж ви справно виконували ваші попередні домашки, то мабуть у вас уже є pagination.html шаблон; таким чином, зможете порівняти ваше рішення із тим, яке ми розробимо пізніше.

Як бачите, нам прийдеться добряче постаратись у нашій в’юшці, щоб надати усю необхідну інформацію для шаблона:

- для навігації по місяцях: prev_month, month_verbose, next_month;
- для шапки таблиці: month_header;

- для посторінкової навігації: page_obj, is_paginated;
- для списку із студентами: students, який матиме дані по студенту та "days" - дані по його щоденному відвідуванню протягом обраного місяця.

Логіка в'юшки

Як і попереднього разу, не будемо одразу пробувати написати усю логіку нашої в'юшки. У багатьох місцях поки приб'ємо статичні дані, в той час як у інших місцях просто залишиком коментарі для подальшого коду.

Оскільки тепер ми використовуємо клас, а не функцію, то і код потрібно класти у інше місце. Як ми вже знаємо із попередніх глав, для збору та підготовки даних для шаблону, Django класи-в'юшки мають метод get_context_data. Ним ми і скористаємось, щоб вкласти усю логіку з підготовки необхідних даних для шаблона. Ось перша дуже проста версія нашої в'юшки:

Перша версія нашої в'юшки

```
1 class JournalView(TemplateView):
2     template_name = 'students/journal.html'
3
4     def get_context_data(self, **kwargs):
5         # get context data from TemplateView class
6         context = super(JournalView, self).get_context_data(**kwargs)
7
8         # перевіряємо чи передали нам місяць в параметрі,
9         # якщо ні - вчисляємо поточний;
10        # поки що ми віддаємо лише поточний:
11        today = datetime.today()
12        month = date(today.year, today.month, 1)
13
14        # обчислюємо поточний рік, попередній і наступний місяці
15        # а поки прибиваємо їх статично:
16        context['prev_month'] = '2014-06-01'
17        context['next_month'] = '2014-08-01'
```

```
18     context['year'] = 2014
19
20     # також поточний місяць;
21     # змінну cur_month ми використовуватимемо пізніше
22     # в пагінації; а month_verbose в
23     # навігації помісячній:
24     context['cur_month'] = '2014-07-01'
25     context['month_verbose'] = u"Липень"
26
27     # тут будемо обчислювати список днів у місяці,
28     # а поки заб'ємо статично:
29     context['month_header'] = [
30         {'day': 1, 'verbose': 'Пн'},
31         {'day': 2, 'verbose': 'Вт'},
32         {'day': 3, 'verbose': 'Ср'},
33         {'day': 4, 'verbose': 'Чт'},
34         {'day': 5, 'verbose': 'Пт'}
35
36     # витягуємо усіх студентів посортированих по
37     queryset = Student.objects.order_by('last_name')
38
39     # це адреса для посту AJAX запиту, як бачите, ми
40     # робитимемо його на цю ж в'юшку; в'юшка журналу
41     # буде і показувати журнал і обслуговувати запити
42     # типу пост на оновлення журналу;
43     update_url = reverse('journal')
44
45     # пробігаємось по усіх студентах і збираємо
46     # необхідні дані:
47     students = []
48     for student in queryset:
49         # TODO: витягуємо журнал для студента i
50         #       вибраного місяця
51
52         # набиваємо дні для студента
```

```
53         days = []
54         for day in range(1, 31):
55             days.append({
56                 'day': day,
57                 'present': True,
58                 'date': date(2014, 7, day).strftime(
59                     '%Y-%m-%d'),
60             })
61
62         # набиваємо усі решту даних студента
63         students.append({
64             'fullname': u'%s %s' % (student.last_name, student.f\
65             irst_name),
66             'days': days,
67             'id': student.id,
68             'update_url': update_url,
69         })
70
71     # застосовуємо піганацію до списку студентів
72     context = paginate(students, 10, self.request, context,
73                         var_name='students')
74
75     # повертаємо оновлений словник із даними
76     return context
```

Як бачите, в кількох місцях ми додали коментарі для подальшого коду, в той час, як у інших місцях набили трохи коду, але поки із статичними даними:

- 4ий рядок: метод get_context_data буде викликаний перед генерацією HTML коду шаблоном; основна його задача - набити даними словник і віддати його далі на обробку; усі нові ключі в словнику будуть доступні в шаблоні;
- 6ий: отримуємо початковий набір даних у словнику від батьківського методу;

- 11ий: в даному блоці ми пізніше додамо код, який обчислюватиме переданий в URL параметрі місяць; а поки ми обчислюємо сьогоднішню дату та місяць (ми усюди прибиватимемо перше число місяця в датах, де нам важливий лише місяць); тут у пригоді нам став вбудований Python модуль `datetime`, який дає нам функцію для оперування датами: `datetime` та `date`;
- 16ий: прибиваємо попередній та наступний місяці у вигляді стрічки в контексті; дані змінні нам потрібні для елементу навігації по місяцях; у цьому ж навігаційному елементі нам потрібно знати поточний рік, тому передаємо і його під ключем ‘`year`’; в наступній ітерації усі дані ми формуватимемо динамічно із запиту;
- 24ий: встановлюємо поки статичні змінні для поточного місяця і його повної назви; змінну ‘`cur_month`’ ми використовуватимемо у віджеті посторінкової навігації, щоб запам’ятовувати місяць журналу під час навігації;
- 29ий: список днів для шапки таблиці; ключ ‘`day`’: номер дня у місяці, ключ ‘`verbose`’: скорочена назва дня у тижні; у наступному кроці також динамізуємо даний список і, звісно, наб’ємо повним списком днів у обраному місяці;
- 37ий: отримуємо усіх студентів з бази посортованих за прізвищем;
- 43ій: цю адресу використовуватимемо для AJAX запиту при оновленні журналу;
- 47ий: у дану змінну збиратимемо усіх студентів із даними для таблиці відвідування;
- 48ий: пробігаємось по усьому списку студентів;
- 49ий: у наступному оновленні нашої в’юшки тут допишемо код, який діставатиме для нас об’єкт журналу із бази для заданого студента та місяця; він нам потрібен, щоб знати поточний стан справ студента по відвідуванню і відобразити його в таблиці;
- 53ій: у змінну `days` якраз і збираємо усі дані студента щодо відвідування протягом заданого місяця;
- 56ий: кожен день у списку це словник, що має: ‘`day`’ - номер дня у місяці, ‘`present`’ - чи був студент присутній того дня, ‘`date`’ - дата цього дня у повному форматі типу “`2014-07-01`”; знову ж таки, поки дані прибили в більшості статично;

- 63ий: маючи усі дні для студента, можемо сформувати словник для самого студента: ‘fullname’ - об’єднуємо ім’я та прізвище студента з допомогою Python інтерполяції стрічок, ‘days’ - щойно приготовані дані з днями місяця, ‘id’ - унікальний ідентифікатор студента в базі, ‘update_url’ - URL адреса обробника запиту на оновлення присутності по студенту;
- 72ий: користуємось функцією paginate, яку пізніше розробимо, щоб уникати дублювання коду у всіх в’юшках, що потребують посторінкової навігації; дана функція прийматиме список елементів, кількість елементів на сторінці, запит, контекст в’юшки та називу змінної, під якою встановлюватиме список елементів у контексті; дана функція буде для нас сама вичитувати параметри навігації із об’єкта запиту та оновлювати контекст;
- 76ий: повертаємо словник контексту.

Як бачите, в’юшка вийшла досить об’ємною. І це при тому, що ми пропустили складніші моменти і лише набили її, в більшості, статичними даними.

Не поспішайте запускати даний код, адже у нас ще бракує імпортів та функції paginate.

Тепер давайте на другому етапі доб’ємо усі місця, де в нас бракує динаміки:

В’юшка журналу: друга половина ;)

```
1 # -*- coding: utf-8 -*-
2
3 from datetime import datetime, date
4 from dateutil.relativedelta import relativedelta
5 from calendar import monthrange, weekday, day_abbr
6
7 from django.core.urlresolvers import reverse
8 from django.views.generic.base import TemplateView
9
10 from ..models import MonthJournal, Student
11 from ..util import paginate
12
```

```
13
14 class JournalView(TemplateView):
15     template_name = 'students/journal.html'
16
17     def get_context_data(self, **kwargs):
18         # get context data from TemplateView class
19         context = super(JournalView, self).get_context_data(**kwargs)
20
21         # check if we need to display some specific month
22         if self.request.GET.get('month'):
23             month = datetime.strptime(self.request.GET['month'], '%Y\
24 -%m-%d'
25             ).date()
26     else:
27         # otherwise just displaying current month data
28         today = datetime.today()
29         month = date(today.year, today.month, 1)
30
31         # calculate current, previous and next month details;
32         # we need this for month navigation element in template
33         next_month = month + relativedelta(months=1)
34         prev_month = month - relativedelta(months=1)
35         context['prev_month'] = prev_month.strftime('%Y-%m-%d')
36         context['next_month'] = next_month.strftime('%Y-%m-%d')
37         context['year'] = month.year
38         context['month_verbose'] = month.strftime('%B')
39
40         # we'll use this variable in students pagination
41         context['cur_month'] = month.strftime('%Y-%m-%d')
42
43         # prepare variable for template to generate
44         # journal table header elements
45         myear, mmonth = month.year, month.month
46         number_of_days = monthrange(myear, mmonth)[1]
47         context['month_header'] = [{ 'day': d,
```

```
48         'verbose': day_abbr[weekday(myyear, mmonth, d)][:2]}
49         for d in range(1, number_of_days+1)]
50
51     # get all students from database
52     queryset = Student.objects.all().order_by('last_name')
53
54     # url to update student presence, for form post
55     update_url = reverse('journal')
56
57     # go over all students and collect data about presence
58     # during selected month
59     students = []
60     for student in queryset:
61         # try to get journal object by month selected
62         # month and current student
63         try:
64             journal = MonthJournal.objects.get(student=student, \
65 date=month)
66         except Exception:
67             journal = None
68
69         # fill in days presence list for current student
70         days = []
71         for day in range(1, number_of_days+1):
72             days.append({
73                 'day': day,
74                 'present': journal and getattr(journal, 'present\\
75 _day%d' % \
76                     day, False) or False,
77                 'date': date(myyear, mmonth, day).strftime(
78                     '%Y-%m-%d'),
79             })
80
81         # prepare metadata for current student
82         students.append({
```

```
83         'fullname': u'%s %s' % (student.last_name, student.first_name),
84     },
85     'days': days,
86     'id': student.id,
87     'update_url': update_url,
88 )
89
90     # apply pagination, 10 students per page
91     context = paginate(students, 10, self.request, context,
92                         var_name='students')
93
94     # finally return updated context
95     # with paginated students
96     return context
```

Цього разу наведений повний текст модуля journal.py разом із імпортами. Що ж ми оновили та додали:

- Зій рядок: імпортуємо вбудовані Python модулі для роботи з датами;
- 4ий: у наступному кроці ми заінсталюємо зовнішній пакет, який значно полегшує роботу із датами; тут ми імпортуємо функцію relativedelta, якою пізніше скористаємося;
- 5ий: для побудови місячного календаря для журналу ми використовуватимемо вбудований Python модуль calendar;
- 11ий: також, у одному із наступних кроків, ми розробимо власну функцію для створення посторінкової навігації; це потрібно для того, щоб уникати дублювання коду у різних в'юшках; тут ми імпортуємо дану функцію для подальшого використання у модулі;
- 22ий: для навігації по місяцях ми передаватимемо в URL адресах параметр “month”; тут ми його пробуємо отримати; якщо він є, тоді створюємо об'єкт datetime з допомогою функції перетворення стрічки у дату по заданому формату: strftime²⁷⁷;

²⁷⁷<https://docs.python.org/2/library/datetime.html#strftime-and-strptime-behavior>

- 28ий: якщо нам не передали дати, тоді показуємо в журналі поточний місяць; для цього створюємо об'єкт дати поточного місяця з числом дня рівним 1;
- 33ій: тут ми вичисляємо попередній і наступний місяці; а робимо це досить просто скориставшись функцією `relativedelta` з пакету `dateutil`; через операції додавання та віднімання ми можемо отак просто посунути дату на один місяць;
- 35ий: передаємо в контекст попередній та наступний місяці; для цього користуємось методом дати `strftime`, який переводить об'єкти у стрічки згідно заданого формату;
- 38ий: аналогічно робимо із назвою місяця; в наступних главах нам прийдеться дану назву місяця перекласти на українську;
- 41ий: також передаємо поточний місяць; але форматуємо його по-іншому, щоб формат підходив для параметра в URL адресах, адже його використовуватимемо в посторінковій навігації пізніше;
- 45ий: запам'ятуємо у змінні номер поточних року та місяця;
- 46ий: отримуємо список днів у обраному місяці; для цього користуємось функцією `monthrange` з модуля `calendar`; їй передаємо номер року та місяця;
- 47ий: передаємо в контекст список днів для заголовку таблиці із відвідуванням; для цього користуємось можливістю динамічно створювати список з допомогою вкладеного циклу (так звані *List Comprehensions* в Python); досить непростий вираз; в циклі ми пробігаємось по списку днів у місяці і запам'ятуємо його під ключем 'day' в словнику; для кожного дня отримуємо його порядковий номер в тижні (з допомогою функції `weekday`), а вже тоді його назву із словника `day_abbr`; отриману назву обрізаємо до двох символів та запам'ятуємо в словнику під ключем 'verbose'; самостійно поекспериментуйте із даним рядком коду, щоб краще зрозуміти як працюють такі функції як `weekday`;
- 63ій: для того, щоб початково набити журнал поточним станом справ, нам потрібно отримати журнал відвідування за вибраний місяць по кожному із студентів; відповідно, в циклі пробуємо отримати об'єкт `MonthJournal` для кожного студента; на рівні полів моделі ми з вами встановили пару студента-місяць як унікальну, тому тут можемо користуватись методом `get`; він видасть помилку лише у випадку, якщо такого

об'єкта поки немає в базі; у такому випадку перехоплюємо помилку і просто присвоюємо змінній `journal` порожнє значення;

- 71ий: пробігаємось по усіх днях місяця; додаємо одиницю до кількості днів, адже функція `range` поверне список днів не включаючи останнього;
- 74ий: кожен день - це словник із даними; ключ 'present' даного словника вказує на те, чи був студент присутній даного дня; атрибут присутності ми витягуємо із об'єкту `journal` через функцію `getattr`; це для того, щоб не отримувати помилки, якщо `journal` є порожнім значенням (див. вище випадок, коли `MonthJournal` ще не встановлений для студента).

Решту рядків модуля залишились практично без змін.

У нашій в'ющі журналу ми використовуємо зовнішній модуль Python під назвою `dateutil`. Його нам потрібно окремо заінсталювати. Для цього оновимо наш файл із списком залежностей проекту `requirements.txt`:

Додаємо `dateutil` до списку залежностей

- 1 `Django==1.7.1`
 - 2 `MySQL-python`
 - 3 `Pillow`
 - 4 `django-crispy-forms==1.4.0`
 - 5 `python-dateutil`
-

і запускаємо ріп з командою інсталяції залежностей:

Не забудьте активувати ваше віртуальне середовище перед запуском команди

- 1 `$ pip install -r requirements.txt`
-

У функції `в'юшки` ми витягуємо і готуємо дані по усіх студентах, проте на сторінці будемо показувати лише 10 із них, а решту будуть доступні з допомогою посторінкової навігації. У високонавантажених проектах така логіка зазвичай оптимізується і інтенсивна робота із даними відбувається лише після того, як елементи уже погруповані по сторінках. Таким чином, ресурси комп'ютера витрачатимуться лише на тих, студентів, які необхідно

показати на поточній сторінці. Оптимізація коду і його швидкодія залишається поза межами даної книги для початківців. Тому залишаю дане завдання вам на домашнє завдання. Насправді, в даному випадку, воно не є уже й таким складним.

I ще й досі наша функція в'юшки не є робочою. Перед тестуванням нам потрібно зробити ще один крок - реалізувати функцію посторінкової навігації по списку та винести HTML код у окремий шаблон.

Оновлюємо посторінкову навігацію

Якщо ви виконували усі завдання попередніх глав книги, тоді у вас посторінкова навігація уже винесена у окремий шаблон. Тут ви лише можете знайти для себе нову ту частину, яка працює із Python кодом. Ну і, звичайно, порівняти ваш варіант із тим, що буде наведений нижче.

Спершу перенесемо код із шаблону у окремий файл:

Створіть новий файл у папці `templates/students` під назвою `pagination.html` і вставте у нього наступний код

```
1  {% if is_paginated %} 
2  {% with order_by=request.GET.order_by reverse=request.GET.reverse %} 
3  <nav> 
4      <ul class="pagination"> 
5          <li> 
6              <a href="?page=1{{if order_by }}&order_by={{ order_by }}{{\ 
7                  endif }}{{ if reverse }}&reverse={{ reverse }}{{ endif }}{{ if \ 
8                  cur_month }}&month={{ cur_month }}{{ endif }}">&laquo;</a> 
9          </li> 
10         <% for p in paginator.page_range %> 
11             <li {{ if object_list.number == p }}class="active"{{ endif }}>
```

```
12      <a href="?page={{ p }}{{if order_by }}&order_by={{ order_b\\
13 y }}{{ endif }}{{ if reverse }}&reverse={{ reverse }}{{ endif }}\\
14 {{ if cur_month }}&month={{ cur_month }}{{ endif }}">{{ p }}</a>
15    </li>
16  {% endfor %}
17  <li>
18    <a href="?page={{ paginator.num_pages }}{{if order_by }}&o\\
19 rder_by={{ order_by }}{{ endif }}{{ if reverse }}&reverse={{ rev\\
20 erse }}{{ endif }}{{ if cur_month }}&month={{ cur_month }}{{ end\\
21 if %}}">
22      &raquo;</a>
23    </li>
24  </ul>
25 </nav>
26 {% endwith %}
27 {% endif %}
```

Як бачите, ми не лише перенесли даний код пагінації, але й дещо оновили його. Тому давайте ще раз пройдемось по основних моментах оновленого шаблону посторінкової навігації:

- 1ий рядок: перевіряємо чи список взагалі потребує навігацію; у випадку, якщо елементів не більше, ніж дозволено на одній сторінці - навігація не потрібна; цю змінну в шаблон ми пізніше передаватимемо із функції pagination, яку згадували раніше в коді в'юшки;
- 2ий: запам'ятуємо змінні order_by та reverse, щоб підтримувати сортування під час посторінкової навігації;
- 3ий: посилання на першу сторінку в навігації; зверніть увагу на те, що ми додали умови на вставляння того чи іншого URL параметру; також ми додали новий параметр в адресу навігації: month; даний параметр беремо із змінної шаблону cur_month; дану змінну ми передаємо з в'юшки журналу, адже важливо, щоб під час посторінкової навігації, зберігався поточний місяць відвідування;

- 10ий: тепер об'єкт класу Paginator доступний глобально в шаблоні напряму, без об'єкта сторінки; його ми пізніше підготуємо у нашій функції pagination;
- 11ий: щоб використовувати наш новий шаблон навігації pagination.html на усіх сторінках, потрібно, щоб змінна, яка містить список елементів була універсальна; необхідно умовою правильного функціонування даної навігації є визначена змінна object_list у шаблоні, який інклудить (включає) даний шаблон pagination.html; якщо пам'ятаєте, кожен інклуд (включення) шаблону pagination.html ми огортали в тег “with”, який визначав змінну object_list в контексті шаблону.

Увесь інший HTML код залишився незмінним.

Тепер, маючи готовий шаблон підготуємо Python функцію, яка буде отримувати дані із запиту, аналізувати їх, розділяти список на групи для навігації та встановлювати дані в шаблонний контекст, які ми їй передаємо. Тобто дана функція буде сама розбиратись із усіма нюансами посторінкової навігації. Для такого роду функцій створимо модуль util.py всередині папки аплікації students:

Модуль для допоміжних функцій util.py

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students
2 $ touch util.py
```

Відкриваємо даний модуль в редакторі і реалізуємо функцію paginate:

Допоміжна функція paginate, модуль util.py

```
1 from django.core.paginator import Paginator, EmptyPage, PageNotAnInt\
2 eger
3
4 def paginate(objects, size, request, context, var_name='object_list'\
5 ):
6     """Paginate objects provided by view.
7
8     This function takes:
9         * list of elements;
10        * number of objects per page;
11        * request object to get url parameters from;
12        * context to set new variables into;
13        * var_name - variable name for list of objects.
14
15     It returns updated context object.
16     """
17     # apply pagination
18     paginator = Paginator(objects, size)
19
20     # try to get page number from request
21     page = request.GET.get('page', '1')
22     try:
23         object_list = paginator.page(page)
24     except PageNotInteger:
25         # if page is not an integer, deliver first page
26         object_list = paginator.page(1)
27     except EmptyPage:
28         # if page is out of range (e.g. 9999),
29         # deliver last page of results
30         object_list = paginator.page(paginator.num_pages)
31
32     # set variables into context
33     context[var_name] = object_list
34     context['is_paginated'] = object_list.has_other_pages()
```

```
35     context['page_obj'] = object_list
36     context['paginator'] = paginator
37
38     return context
```

Основну логіку ви вже бачили у попередніх главах, коли ми з вами реалізували список студентів. Але тут є кілька додаткових моментів по роботі із контекстом, а також можливістю додаткової конфігурації:

- 4ий рядок: визначаємо функцію paginate; дана функція приймає список елементів, кількість елементів на одній сторінці, об'єкт запиту (з нього ми отримуватимемо дані про поточний стан навігації), контекст (сюди ми додамо необхідний набір даних для подальшого використання в шаблоні pagination.html), назва змінної, під якою встановлювати список готових для навігації елементів (по замовчуванню змінна називається object_list);
- 5ий: документаційна стрічка функції, яка описує призначення функції, її аргументи та те, що функція повертає;
- 17ий: починаючи із даного рядка і закінчуючи рядком 30, маємо увесь код скопійований із в'юшки списку студентів; ми цей код перенесли і виокремили у функцію; весь цей кусок коду ви уже бачили і мабуть розібрались із усіма деталями;
- 33ій: встановлюємо змінну - список елементів готовий для навігації на поточній сторінці;
- 34ий: встановлюємо змінну для шаблону is_paginated; це булеванівська змінна, яка вказує на те, чи потрібна навігація; якщо елементів у списку не більше, ніж size, тоді немає змісту відображати посторінкову навігацію;
- 35ий: на всякий випадок також передаємо в шаблон об'єкт сторінки під стандартною назвою page_obj;
- 36ий: об'єкт класу Paginator також передаємо в контекст шаблона; він нам пригодиться;
- 38ий: наприкінці повертаємо оновлений контекст.

Основний шаблон журналу уже готовий до використання оновленої пагінації, тому можемо сміливо оновляти сторінку Відвідування та насолоджуватись плодами нашої праці. Спробуйте понавігувати між місяцями, а також між сторінками студентів. Зауважте, що навігація по списку студентів запам'ятовує поточноЭ обраний місяць. Це ми з вами постарались!

На домашнє завдання залишаю вам переробити сторінки із списками студентів, груп, екзаменів (якщо вони ще не оновлені) так, щоб вони працювали з використанням нашої функції `paginate` та шаблона `pagination.html`. Це зекономить кілька десятків дублюючого коду.

На даний момент ми з вами реалізували досить нетривіальну логіку для в'юшки відвідування. Ще можуть залишатись деякі моменти, що не повністю є зрозумілими. Якщо більшість речей під питанням, тоді рекомендую ще раз повільно пройтись по даній секції глави. Якщо ж є лише дрібні запитання, тоді пропоную продовжувати наш шлях, а пізніше повернутись знову сюди. Є шанси, що пізніше усі деталі складутися у єдину логічну картину! Просто вашій підсвідомості може бути потрібно трохи більше часу, щоб все осягнути і розкласти по поличках.

Стилі таблиці

На завершення нашої непростої роботи над візуальною частиною закладки відвідування зробимо ще одне невеличке покращення: вирівняємо відстані навколо чекбоксів та записів днів у заголовку таблиці. На даний момент таблиця занадто розтягнута та клітинки із чекбоксами не є квадратними, що додало б крашного вигляду та зручності в користуванні журналом.

		← January 2015 →																							
#	Студент	Th 1	Fr 2	Sa 3	Su 4	Mo 5	Tu 6	We 7	Th 8	Fr 9	Sa 10	Su 11	Mo 12	Tu 13	We 14	Th 15	Fr 16	Sa 17	Su 18	Mo 19	Tu 20	We 21	Th 22	Fr 23	Sa 24
1	Podoba Vitaliy	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>							
2	Микола Біндас	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
3	Danylo Лінтер	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
4	Іздрік Ярослав	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

Таблиця із розтягнутими клітинками

Для цього відкриємо наш старий добрий main.css файл із стилями і додамо наступні правила:

Вирівнюємо комірки з чекбоксами та заголовки таблиці

```

1 /* Journal Table Styles */
2 td.day-box,
3 th.day-header {
4   padding: 4px !important;
5   vertical-align: middle !important;
6   text-align: center;
7 }
```

При написанні HTML коду для нашої сторінки відвідування ми додали клас “day-box” до комірок із чекбоксами, а також клас “day-header” до комірок в шапці таблиці. У вищепереліченному коді CSS ми скористалися даними класами, щоб прив’язати додаткові стилі до комірок таблиці:

- 4ий рядок: обмежуємо внутрішню відстань всередині комірки до 4 пікселів із кожної із сторін; зауважте декларацію `!important`²⁷⁸ - вона для того, щоб перекрити існуючі стилі, що приходять із Twitter Bootstrap без використання занадто складних CSS селекторів; дана декларація дозволяє знехтувати каскадністю стилів;
- 5ий: центруємо елемент комірки вертикально;

²⁷⁸<http://htmlbook.ru/css/!important>

- бий: центруємо елемент комірки горизонтально.

Тепер можемо остаточно вважати, що робота по візуальній частині завершена!

...

В даній секції ми з вами скористались трохи іншим підходом при розробці функціоналу, ніж зазвичай використовує початківець. Початківець пише невеликий кусок коду і намагається якомога швидше його запустити і протестувати. І такими маленькими кроками повільно доходить до кінцевого результату. Саме таким чином ми з вами намагались працювати у попередніх главах.

Проте для закладки Відвідування ми спробували варіант, якого, зазвичай, притримуються професійні програмісти: реалізують майже весь необхідний функціонал і вже тоді пробують це все запустити. В момент запуску може виникати багато помилок і неточностей, які потрібно “допилювати”. Чим краще програміст розбирається в поточних технологіях, тим менше спроб йому знадобиться для запуску усього того коду, який він запрограмував протягом останніх кількох годин.

Кожен програміст знаходить баланс між даними двома підходами взалежності від поточного завдання. Якщо подібне завдання вже було у його практиці, то швидше за все, він зможе запустити кінцевий код з першої спроби.

Обробник змін в журналі

Ми з вами добряче постарались, щоб заставити працювати наш журнал відвідування у режимі перегляду. Це була основна частина роботи. Нам залишилась, хоч і не така трудоємка, але не менш важлива частина - можливість змінювати даний журнал.

Для цього нам потрібно буде додати власний Javascript код, який відслідковуватиме кліки по чекбоксах і робитиме запит на сервер для збереження відповідних змін в журналі.

Тому почнемо із створення інфраструктури для кастомного Javascript коду. Створимо папку js всередині папки із статичними ресурсами, а всередині неї новий файл під назвою main.js. В ньому будеувесь наш Javascript код по проекту:

Додаємо main.js файл

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students/static
2 $ mkdir js
3 $ touch js/main.js
```

Тепер включимо наш новостворений файл у base.html шаблон, щоб він був доступний на кожній сторінці нашої аплікації:

Додаємо наш власний Javascript файл до base.html шаблону

```
1 <!-- Javascripts Inclusion -->
2 <script src="http://ajax.googleapis.com/ajax/libs/jquery/2.1.1/jqu\ 
3 ery.min.js"></script>
4 <script src="https://cdn.jsdelivr.net/bootstrap/3.3.0/js/bootstrap\ 
5 .min.js"></script>
6 <script src="{% static "js/main.js" %}"></script>
7 {% block extra_js %}{% endblock extra_js %}
```

На передостанньому рядку можемо бачити вкладення Javascript файлу main.js. Важливо, що ми включили його після jQuery скрипта, адже наш код активно використовуватиме дану бібліотеку.

Відкриваємо файл main.js у своєму редакторі і набиваємо базовою структурою:

Використовуючи jQuery запускаємо тестову функцію на клік по чекбоксу

```
1 function initJournal() {
2     $('.day-box input[type="checkbox"]').click(function(event){
3         alert('test');
4     });
5 }
6
7 $(document).ready(function(){
8     initJournal();
9 });
```

В даному варіанті наш скрипт чіпляє функцію-обробник до усіх чекбоксів. Дану функція показуватиме вікно із словом ‘test’ при кожному кліку. А тепер давайте порядково розберемось:

- 1ий рядок: декларуємо функцію під назвою `initJournal`; дана функція чіплятиме Javascript обробники на елементи управління журналом; `init` частина в назві функції я закладаю як скорочення від `initialize`, тобто функція, яка відповідає за ініціалізацію функціоналу журналу;
- 2ий: працюємо з бібліотекою `jQuery` через функцію під назвою “`$`”; основне її завдання: повернати елементи поточної сторінки за заданим селектором²⁷⁹; у нашому випадку ми шукаємо усі чекбокси в таблиці журналу; кожну комірку ми промаркували класом “`day-box`”, тому до потрібних чекбоксів можемо доступитись через селектор `.day-box input=[type="checkbox"]`: тег `input` з атрибутом `type` рівний значенню `checkbox`; функція “`$`” поверне нам список усіх чекбоксів в журналі і до кожного елементу в даному списку ми додаємо обробник події `кліку мишкої`²⁸⁰; як бачите, функція “`$`” повертає не прості об’єкти із об’єктної моделі документа (DOM), а спеціально огорнуті бібліотекою `jQuery` об’єкти з масою додаткових атрибутів та властивостей; серед них є і можливість навішування подій; в нашому випадку це метод `click`; єдиним аргументом ми передаємо функцію; дана функція спрацьовуватиме при кліку на чекбоксі;
- 3ий: функція `alert`²⁸¹ виводить модальне вікно із переданим їй повідомленням;
- 4ий: подвійні дужки; перші, фігурні, закривають визначення функції-обробника кліку; другі, круглі, закривають виклик методу `click`; зауважте, що після круглих дужок в Javascript обов’язково повинна йти крапка з комою;
- 7ий: об’єкт документа в `jQuery` має метод `ready`; даному методу ми передаємо функцію, яка буде запущена при повному завантаженню і генерації об’єктної моделі документа (DOM); весь код всередині переданої `ready` методу функції буде запущеним лише після завантаження DOM;

²⁷⁹<http://bit.ly/vpjqsels>

²⁸⁰<http://bit.ly/vpjqlclick>

²⁸¹<http://javascript.ru/alert>

- 8ий: коли готовий DOM, лише тоді запускаємо нашу функцію; в протилежному випадку браузер може підтягнути наш Javascript файл із сервера та запустити його ще до того, як необхідні нам елементи на сторінці будуть доступними для роботи з ними; в такому випадку наш код завершиться із помилкою.

Селектори в jQuery працюють так само, як і селектори в мові стилів CSS. jQuery підтримує масу додаткових селекторів, які не є доступними в CSS.

Спробуйте перезавантажити вашу сторінку відвідування та клацнути у будь-якому із чекбоксів. Якщо все правильно зробили, то повинні отримати вікно з повідомленням ‘test’.

Таким чином, маємо заготовку і все необхідне, щоб додати логіки на клієнтській стороні. Наша функція обробник повинна збирати дані про клік, відсиляти їх на сервер, отримувати та обробляти відповідь належним чином. Для відправки запиту на сервер скористаємося jQuery функцією [ajax²⁸²](#).

Ось як виглядатиме оновлена функція initJournal:

Відправляємо AJAX запит на сервер

```

1 function initJournal() {
2     $('.day-box input[type="checkbox"]').click(function(event){
3         var box = $(this);
4         $.ajax(box.data('url'), {
5             'type': 'POST',
6             'async': true,
7             'dataType': 'json',
8             'data': {
9                 'pk': box.data('student-id'),
10                'date': box.data('date'),
11                'present': box.is(':checked') ? '1' : '',
12            }
13        })
14    })
15 }

```

²⁸²<http://bit.ly/vpajax>

```
12     'csrfmiddlewaretoken': $('input[name="csrfmiddlewaretoken"]') \
13   ).val()
14   },
15   'error': function(xhr, status, error){
16     alert(error);
17   },
18   'success': function(data, status, xhr){
19     alert(data['key']);
20   }
21 });
22 });
23 }
```

Давайте пройдемось по кожному рядку, щоб краще зрозуміти як працює запит на сервер:

- Зій рядок: визначаємо змінну `box`, яка зберігатиме для нас посилання на поточний чекбокс, по якому клікнули; змінна “`this`” в мові Javascript відіграє подібну роль, як “`self`” в мові Python - зсилається на поточний об’єкт даного класу; проте у функціях-обробниках подій, з певною обгортою від бібліотеки jQuery, “`this`” вказує на елемент, над яким відбулась подія; у нашому випадку це елемент `input` типу `checkbox`; далі в коді ми неодноразово звертатимемось до змінної `box`;
- 4ий: функція `ajax`²⁸³ виконує запит на сервер; зауважте, що її ми отримуємо через змінну “`$`” з допомогою позначення крапки `(.)`, що працює таким самим чином як в мові Python: доступ до атрибутів та методів об’єкта; в дану функцію ми передаємо два аргументи: адресу обробника запиту на сервері, словнико-подібний набір налаштувань запиту; адресу для запиту на сервер ми отримуємо із попередньо встановленого атрибута на кожному із чекбоксів - “`data-url`”; jQuery бібліотека дає можливість доступатись до таких атрибутів через спеціальний метод `data`; таким чином, в аргумент даного методу передаємо назvu атрибута без префікса

²⁸³<http://api.jquery.com/jquery.ajax/>

“data-”; завдяки тому, що box ми “одягнули” в jQuery обортку з допомогою “\$()” функції, даний об’єкт володіє усіма додатковими методами та атрибутами, що приходять із даною бібліотекою; включаючи метод data;

- 5ий: починаємо список додаткових налаштувань для AJAX запиту; type - це тип запиту на сервер; в даному випадку запит робить зміни в базу даних, тому важливо виконувати даний запит методом POST;
- 6ий: параметр asymp вказує на те, чи відсылати запит у асинхронному режимі; це означає, що наступний рядок коду після виклику аjax функції буде запущеним одразу, не очікуючи на відповідь від сервера;
- 7ий: параметр dataType вказує на формат даних, що мають прийти із сервера; ми використовуватимемо JSON; крім того, є також можливість отримувати HTML, XML та Javascript код;
- 8ий: під ключем data передаємо набір параметрів, що підуть в тіло поста; тобто це ті змінні, які будуть доступні на серверній стороні через словник request.POST;
- 9ий: аналогічним чином як ми витягували “data-url” параметр з тегу чекбокса, тут витягуємо і передаємо ID студента; передаємо його під ключем “pk”;
- 10ий: передаємо на сервер дату поточного чекбокса;
- 11ий: перевіряємо чи даний клік по чекбоксу поставив чи забрав галочку; відповідно, якщо поставив галочку - відсылаємо на сервер під ключем “present” значення “1”, в протилежному випадку - порожню стрічку; метод `is284` допомагає визначити чи даний об’єкт задоволяє селектору “:checked”; “:checked” вказує на те, чи поточний елемент є вибраним; в даному виразі використана цікава конструкція умови: [умова] ? [результат якщо правдива умова] : [результат якщо неправдива умова]; вираз перед знаком питання служить в якості умови; вираз після знака питання - в якості результату, якщо умова повертає true; вираз після двокрапки - в якості результату, якщо умова повертає false;
- 12ий: оскільки тип нашого AJAX запиту POST, маємо обов’язково передати параметр під назвою ‘csrfmiddlewaretoken’; він передасть унікальне згенероване число для перевірки на серверній стороні і підтвердження того, що наш запит зроблений дійсно із сторінки Відвідування; раніше

²⁸⁴[http://jquerybook.ru/api/.is\(\)-fn126.html](http://jquerybook.ru/api/.is()-fn126.html)

в нашому шаблоні ми вставили тер ‘csrf’, тому тепер його просто шукаємо і повертаємо значення відповідного прихованого поля на сторінці; метод val повертає значення поля форми, яке ми знайшли з допомогою селектора ‘input[name=”csrfmiddlewaretoken”]’;

- 15ий: під ключем ‘error’ передаємо функцію-обробник для випадку, коли на сервері стала помилка під час обробки нашого запиту; дана функція приймає об’єкт запиту (xhr - XML Http Request), статус відповіді із сервера та саму помилку;
- 16ий: код даної функції доволі простий; передану помилку відображаємо у модальному браузерному вікні з допомогою вбудованої функції alert; у складніших випадках можна показувати дане повідомлення як статусне повідомлення на сторінці; можете спробувати зробити подібного роду обробник помилки самостійно;
- 18ий: під ключем ‘success’ передаємо функцію-обробник випадку, коли на сервері наш запит успішно обробили і відправили відповідь; дана функція приймає дані із серверу, статус відповіді та об’єкт запиту;
- 19ий: поки залишимо код цієї функції дуже простим: показуватимемо дані, що прийшли із сервера у модальному вікні браузера; як зазначено вище, дані із сервера ми приймаємо у форматі JSON, тому можемо доступатись до конкретних даних через заготовані на сервері ключі; нехай це буде поки єдиний ключ під іменем ‘key’; далі ми ускладнимо дану функцію необхідною логікою, щоб користувач розумів, що зміни на сервері відбулись успішно;
- 20ий: зауважте, що ми не ставили коми після останнього елемента ключів словника; деякі браузери можуть таку кому протрактувати як синтаксичну помилку; в такому випадку велика частина Javascript коду на сторінці просто не спрацює.

Функція `jQuery.ajax` виконує для нас не лише запит на сервер, але й забирає від нас усі проблеми щодо відмінностей між реалізаціями об’єкта запиту у різних браузерах. Також дана функція дає нам можливість навішувати додаткові обробники при отриманні відповіді від сервера значно простішим

інтерфейсом (параметри success, error і тому подібні).

Символ “\$”, який приходить з бібліотекою jQuery працює і як функція (у цьому випадку повертає об'єкти сторінки по переданому їй селектору), і як об'єкт, через який можемо доступатись до усього багато функціоналу цієї бібліотеки.

Спробуйте тепер перезавантажити вашу сторінку і поклацати по чекбоксах. Повинні отримати помилку в модальному вікні з повідомленням: Method not allowed. Це тому, що ми поки не реалізували обробника на серверній стороні для нашого запиту.

Якщо ж нічого не змінилось, тоді, швидше за все, ваш браузер закешував попередній стан файлка main.js. У цьому випадку допоможе комбінація клавіш Ctrl-Shift-R. Це так званий форс-релоад, коли ми заставляємо браузер наново перетягнути статичні файли із сервера. При деплойментах на продакшин зазвичай статичні файли перейменовують, щоб існуючі користувачі одразу отримали оновлений функціонал. Адже ніхто із них не буде робити форс-релоад будучи на вашому сайті. Все залежить від налаштувань кешування фронт-енд ресурсів вашої аплікації, яке є поза межами даної книги.

Обробник змін на сервері

Ми практично завершили із частиною функціоналу на фронт-енд стороні сторінки Відвідування. Час перейти на серверну сторону і додати обробник нашого AJAX запиту для збереження змін в журналі.

Для початку зробимо заготовку для функції обробника нашого AJAX запиту і зробимо так, щоб повідомлення про недозволений метод зникло. Для цього додамо новий метод до класу JournalView. Як ви знаєте, TemplateView клас

надає нам метод `dispatch`, який розкидає запити на різні методи класу в'юшки залежності від типу запиту. Відповідно, для нашого AJAX запиту нам потрібно реалізувати новий метод під назвою `post`. На нього попадатимуть усі запити типу POST, які йдуть на адресу сторінки Відвідування.

Таким чином, ніяких змін в URL шаблонах ми не потребуємо. Усе, що потрібно - це додати новий метод у клас:

Метод `post` у класі в'юшки журналу, всередині `JurnalView`

```
1 ...
2 from django.http import JsonResponse
3 ...
4
5 class JurnalView(TemplateView):
6
7     ...
8
9     def post(self, request, *args, **kwargs):
10        return JsonResponse({'key': 'value'})
```

Для формування відповіді із сервера у форматі JSON ми скористаємось класом `JsonResponse`. Він для нас встановить необхідний заголовок “Content-Type”, щоб наш код на клієнті знати, якого формату прийшли дані. Також даний клас правильно відформатує передані йому дані.

Сам метод `post` приймає першим параметром об'єкт запиту (ключове слово `self` ми не враховуємо), а також усі передані йому в URL адресі параметри через URL шаблон. Додаткові дані попадають або в список `args`, або в словник `kwargs`, залежності від того, яким саме чином були передані в дану функцію.

Даний метод повертає об'єкт типу `JsonResponse`. При створення цього об'єкту ми йому передаємо словник із даними. В нашему випадку це простий словник із одним ключем ‘`key`’ та статичною стрічкою ‘`value`’ в якості значення. Саме цей ключ потрібен для правильного функціонування нашого ‘`success`’ колбека (callback з англ. передзвонити, запрошення на зворотній зв'язок) в Javascript коді.

Спробуйте тепер покласти по чекбоксах. На кожен клік повинні отримувати модальне вікно із повідомленням ‘value’.

Давайте зробимо нарешті щось корисне всередині серверного методу post. Ось алгоритм дій для нашого методу:

- отримати дані із запиту про студента, дату та присутність студента;
 - конвертувати отриману дату в об'єкт Python дати для подальшої роботи з нею;
 - отримати з бази студента за даною ID студента;
 - отримати з бази журнал за поточний місяць та для переданого нам студента;
 - якщо такого журналу поки не існує - створити його;
 - встановити нове значення в атрибут поля журналу, що відповідає за обраний день;
 - зберегти зміни в базу;
 - повернути JsonResponse об'єкт із статусом, що все завершилось добре.

Ось як виглядатиме повний код оновленого методу:

Обробка поста журналу на сервері

```
15      # set new presence on journal for given student and save res \
16  ult
17      setattr(journal, 'present_day%d' % current_date.day, present)
18  journal.save()
19
20      # return success status
21  return JsonResponse({'status': 'success'})
```

Давайте детально розберемо кожен рядок:

- 2ий рядок: запам'ятуємо словник POST в окрему змінну; це зекономить нам трохи коду, при наступних зверненнях до даних поста;
- 5ий: з допомогою функції `strptime`²⁸⁵ перетворюємо стрічку дати у об'єкт дати; дана функція першим аргументом приймає стрічку, що означає дату, а другим аргументом формат, в якому передана дата; запам'ятуємо результат в змінну `current_date`; функція `strptime` повертає об'єкт дати разом з часом; оскільки час нам не потрібен, обрізаемо його з допомогою методу `date`;
- 7ий: в змінну `month` запам'ятуємо дату, що рівна першому дню обраного року та місяця; дана змінна пригодиться нам при пошуку об'єкту журналу для обраного місяця; адже, якщо пам'ятаєте, усі наші дати в журналі містять дату першого місяця; день нам не важливий, тому ми прив'язались до першого дня місяця;
- 8ий: в даному рядку ми скористались умовною конструкцією подібною до тої, яку бачили напередодні в Javascript коді; але Python немає спеціальних операторів для таких скорочених умов; натомість користуємося його власними логічними операторами; властивість логічних виразів в мові Python полягає у тому, що вони не просто повертають булеванівську змінну, але будь-що що ми передамо в якості кінцевих операндів даних операторів; якщо значення 'present' непорожнє, тоді вираз поверне True, в протилежному випадку False;
- 9ий: маючи ID студента, можемо з легкістю отримати об'єкт студента з бази даних;

²⁸⁵<https://docs.python.org/2/library/datetime.html#datetime.datetime.strptime>

- 12ий: тепер, маючи об'єкт студента та підготовану дату необхідного місяця, пробуємо витягнути з бази об'єкт журналу; для цього використовуємо метод `get_or_create286`; якщо потрібного об'єкта не буде в базі, то ми не отримаємо помилки, натомість Django створить новий об'єкт і поверне його нам; даний метод повертає список із двох елементів: сам об'єкт з бази даних та булеванівську змінну, яка дає нам знати, чи об'єкт був знайдений в базі, а чи новостворений;
- 17ий: встановлюємо атрибут присутності у потрібне значення; для цього ми не використовуємо позначення крапки для доступу до атрибуту, а скористались функцією `setattr`, адже назва атрибуту динамічно формується взалежності від дня місяця: “`‘present_day%d’ % current_date.day`”;
- 18ий: зберігаємо оновлений об'єкт журналу в базу даних;
- 21ий: повертаємо `JsonResponse` із ключем ‘status’ встановлений у значення ‘success’.

Таким чином, тепер кожен AJAX запит на сервер закінчуватиметься змінами у базу даних. Спробуйте покласти тепер по чекбоксах. На кожен клік повинні отримувати модальне вікно із повідомленням ‘success’. Також після перезавантаження сторінки Відвідування усі чекбокси повинні зберегти зміни, які ви зробили напередодні.

Якщо ж отримуєте повідомлення про помилку, значить десь на сервері недопрацьований метод `post`. Спробуйте глянути і розібратись у проблемі. Можна навіть з використанням Python `pdb` дебагера. Як ним користуватись ви уже могли бачити з відео уроків, що йдуть разом з Рекомендованим пакетом даної книги.

Візуальний фідбек

Зрозуміло, що кожного разу отримувати таке модальне вікно при кліку в чекбоксі є досить незручно. Натомість можна його або просто забрати, або замість нього придумати кращий спосіб показувати користувачу, що при кожному кліку щось насправді відбувається і його зміни зберігаються на сервері.

²⁸⁶<http://djbook.ru/rel1.4/ref/models/querysets.html#get-or-create>

З англ. feedback - відповідь, зворотній зв'язок.

Тому далі зробимо невеликий візуальний фідбек для дій користувача. Оскільки ми не маємо повноцінної форми з кнопками та набором полів, а запит відбувається при кожному кліку на чекбокс, робити систему оповіщення, яка блокуватиме інші дії при кожному кліку, буде перебором.

У нашому випадку буде достатньо реалізувати статусне повідомлення, яке регулярно оновлювати при кожному кліку чекбокса.

Давайте вставимо початкове повідомлення, яке пояснить користувачу, що кожен клік по чекбоксу одразу записується в базу. Таким чином одним пострілом вб'ємо два зайці:

- проінструктуємо користувача як працювати із сторінкою;
- приб'ємо місце для подальших AJAX нотифікацій; адже якщо вставляти статусне повідомлення без перезавантаження сторінки, весь контент нижче нього зсуватиметься створюючи ефект стрибка; а це вже перериває процес ведення журналу викладачем.

То ж додамо статусне повідомлення до шаблону журналу:

Наведено лише частину шаблону журналу, де працюємо із статусними повідомленнями

```
1 ...
2
3 {% if not students %}
4
5 <div class="alert alert-warning">
6     Поки немає жодного студента в базі. Будь-ласка,
7     <a href="{% url "students_add" %}">додайте</a> першого.
8 </div>
9
10 {% else %}
```

```
11
12 <div class="alert alert-warning">
13     Зміни в журнал зберігаються автоматично при кожному кліку в кліт\
14     інці
15     таблиці.
16     <span id="ajax-progress-indicator">Йде збереження...</span>
17 </div>
18
19 <p id="journal-nav" class="text-center">
20     <a href="?month={{ prev_month }}"
21         title="Попередній Місяць">&larr; </a>
22     <!-- TODO: translate month name -->
23     <strong>{{ month_verbose }} {{ year }}</strong>
24     <a href="?month={{ next_month }}"
25         title="Наступний Місяць">&rarr; </a>
26 </p>
27
28 ...
```

Що ж ми змінили:

- 12ий рядок: в гілці “else”, коли уж маємо студентів у списку, додаємо ще одне статусне повідомлення, яке завжди буде відображатись; воно пояснює користувачу, що його кліки по чекбоксах автоматично зберігаються в базу;
- 16ий: всередині статусного повідомлення вставляємо елемент з ID “ajax-progress-indicator”, який використовуватимемо в Javascript коді, щоб показувати, що на фоні відбувається запис в базу (тобто запит на сервер); ми його початково сховаемо, а при кожному запиті покажуватимемо, щоб користувач бачив візуальний фідбек своїх дій; це буде зовсім простий механізм і в складніших формах, зазвичай, реалізують більш багаті візуальні ефекти і функціонал для обробки форми.

Зрозуміло, що текст “Йде збереження...” не актуальний на початку завантаження нашої сторінки, тому початково потрібно його заховати. Зробимо це

з допомогою стилів. Відкриваємо наш файл із стилями main.css і додаємо наступний кусок коду:

Ховаємо елемент-індикатор AJAX запитів, main.css

```
1 #ajax-progress-indicator {  
2   display: none;  
3 }
```

Думаю, на даний момент, все досить зрозуміло. Елемент під ID “ajax-progress-indicator” ховаємо з допомогою властивості display.

Знову ж таки, якщо зміни в стилях не застосовуються після перевантаження сторінки, спробуйте форс-релоад варіант.

Бракує найголовнішого - оновленого Javascript коду, який показуватиме та ховатиме наше повідомлення “Йде збереження...” у потрібний момент.

Хоч змін і небагато, проте вони торкаються різних місць функції initJournal. Тому наводжу повний код даної функції в оновленому стані:

Показуємо та ховаємо AJAX індикатор у правильні моменти

```
1 function initJournal() {  
2   var indicator = $('#ajax-progress-indicator');  
3  
4   $('.day-box input[type="checkbox"]').click(function(event){  
5     var box = $(this);  
6     $.ajax(box.data('url'), {  
7       'type': 'POST',  
8       'async': true,  
9       'dataType': 'json',  
10      'data': {  
11        'pk': box.data('student-id'),  
12        'date': box.data('date'),  
13      }  
14    }).done(function(data){  
15      if(data['status']){  
16        indicator.show();  
17      }  
18    }).fail(function(error){  
19      console.log(error);  
20    });  
21  });  
22 }  
23  
24 $(document).ready(function(){  
25   initJournal();  
26 });
```

```
13     'present': box.is(':checked') ? '1': '',
14     'csrfmiddlewaretoken': $('input[name="csrfmiddlewaretoken"]') \
15   ).val()
16   },
17   'beforeSend': function(xhr, settings){
18     indicator.show();
19   },
20   'error': function(xhr, status, error){
21     alert(error);
22     indicator.hide();
23   },
24   'success': function(data, status, xhr){
25     indicator.hide();
26   }
27 });
28 });
29 }
```

Ми додали кілька нових рядків до функції initJournal:

- 2ий рядок: шукаємо на сторінці елемент, що містить наш індикатор AJAX запиту та запам'ятуємо його у змінну indicator; операції з елементами на сторінці та їх пошук є доволі ресурсозатратними, тому один раз витягуємо даний елемент на сторінці і зберігаємо його в змінну для подальшого використання;
- 17ий: функція аjax також має налаштування під назвою ‘beforeSend’; сюди можна передати функцію, яка буде викликатись безпосередньо перед запуском запиту на сервер; дана функція отримує об'єкт запиту та налаштування аjax функції; якщо ми повернемо false із даної функції, то AJAX запит буде відмінено; в нашому випадку дана можливість нам не потрібна;
- 18ий: перед початком запиту на сервер ми показуємо повідомлення “Йде збереження...”; jQuery бібліотека наділила наш об'єкт індикатора методами show і hide, які показують та ховають елемент сторінки, відповідно;

- 22ий: до функції, що спрацьовуватиме у випадку помилки, ми додали рядок, який ховає наш індикатор; ідея індикатора в тому, щоб його показувати протягом роботи AJAX запиту, тому після того як отримали відповідь із сервера (помилкову чи успішну), ховаємо дане повідомлення;
- 25ий: аналогічним чином ховаємо індикатор запиту після успішного AJAX запиту.

Знову перевантажте вашу сторінку у веб-переглядачі (бажано із форс-релоадом) та потестуйте результат. Під час кліку, на невеликий проміжок часу (долі секунди), буде показуватись стрічка “Йде збереження...”. На локальному розробницькому сервері запит на сервер відбувається на стільки швидко, що інколи важко зауважити дану стрічку. Проте на продакшин сервері шлях до сервера від браузера буде значно більший, і, відповідно, дане повідомлення користувач зможе з легкістю зауважувати.

Візуальний фідбек AJAX запитів

В якості вправи рекомендую покращити нашу функцію-обробник помилкової відповіді із сервера. Таким чином, щоб вона повертала деталі помилки у статусне повідомлення під спеціальним червоним кольором. Гляньте Twitter Bootstrap алерти для “error” статусів.

Журнал студента

На даний момент ми повністю реалізували закладку Відвідування. Але, якщо пам'ятаєте, ще на самому початку ми з вами додали меню Відвідування

для кожного студента в списку на сторінці Студенти. Даний елемент меню повинен відправляти користувача на сторінку відвідування поточно обраного студента. В даній секції реалізуємо цей функціонал.

На перший погляд завдання виглядає як досить складне, але за кілька хвилин побачити як легко і швидко ми реалізуємо дану функцію.

Давайте спочатку наведемо список змін необхідних для реалізації даного завдання:

- оновити модуль urls.py, щоб адреса типу “/journal/[student_id]” запускала нашу в’юшку журналу і при цьому передавала ідентифікатор студента як параметр;
- в шаблоні із списком студентів оновити посилання на елемент меню Відвідування;
- оновити в’юшку журналу, щоб вона могла відображати не лише усіх студентів, але й приймала параметр - ID студента і відображала лише його; таким чином в’юшка працюватиме в обидвох режимах.

Відповідно почнемо із оновлення URL шаблонів. Наше завдання зробити адресу типу “/journal/3” відображати журнал відвідування для студента під ідентифікатором 3:

Ускладнюємо шаблон журналу додатковою компонентою наприкінці регулярного виразу, urls.py

```
1   url(r'^journal/(?P<pk>\d+)?/?$', JournalView.as_view(), name='journal'),
```

Замість того, щоб додавати новий URL шаблон для адрес Відвідування одного студента, ми оновили існуючий шаблон журналу і зробили так, щоб він працював і для закладки Відвідування і для відвідування одного студента. Зробили це доволі просто, додавши іменовану групу, яка відловлює ID студента, а також слеш наприкінці виразу. Після обидвох компонент ми поставили метасимвол - знак запитання. Він робить обидві цих компоненти опціональними (необов’язковими). Таким чином одним каменем вбили дві пташки.

Далі, уже маючи готовий URL шаблон, скористаємося ним і оновимо шаблон списку студентів:

Оновлюємо адресу посилання в меню студентів, students_list.html

```

1      <ul class="dropdown-menu" role="menu">
2          <li><a href="{% url "students_edit" student.id %}">Редаг\ 
3  увати</a></li>
4          <li><a href="{% url "journal" student.id %}">Відвідуванн\ 
5  я</a></li>
6          <li><a href="{% url "students_delete" student.id %}">Вид\ 
7  алити</a></li>
8      </ul>

```

Рядок, який ми оновили є рядок номер 4. Замість статично прибитої адреси на ‘/journal’ сторінку тепер ми використовуємо тер url і шаблон під назвою journal. Як бачите, даний шаблон можемо використовувати як самостійно, так і з переданим додатковим аргументом - ID студента.

Залишився останній, основний крок: оновити логіку в’юшки журналу, щоб вона працювала у двох режимах: виводила список студентів, а також виводила єдиного студента, якщо був переданий параметр ‘pk’.

Оновимо код в’юшки, а саме її метод get_context_data. В тому місці, де ми формували змінну queryset (спісок усіх студентів), додамо більше логіки:

Віддаємо обраного студента

```

50      # get all students from database, or just one if we need to
51      # display journal for one student
52      if kwargs.get('pk'):
53          queryset = [Student.objects.get(pk=kwargs['pk'])]
54      else:
55          queryset = Student.objects.all().order_by('last_name')

```

Давайте детальніше глянемо на оновлену логіку:

- 52ий рядок: наш метод приймає усі аргументи у вигляді словника kwargs; відповідно і додатковий аргумент ‘pk’ (ID студента), якщо прийде, то буде знаходитись всередині даного словника; тому в даному рядку перевіряємо чи отримали ID студента в URL адресі;

- 53ій: якщо так, тоді отримуємо з бази єдиного студента і огортаємо його в список одного елемента; це потрібно для того, щоб змінна queryset і надалі була списком; це дозволить не змінювати іншої частини коду методу `get_context_data`;
- 55ий: якщо ID студента нам не передали (тобто потрібно відобразити закладку Відвідування), тоді queryset змінна залишається такою ж, якою була до цього моменту - список усіх студентів посортованих по прізвищу.

Ось так, 4-ма стрічками коду, ми адаптували нашу в'юшку, щоб працювала для обидвох схем: закладка Відвідування та журнал відвідування для єдиного студента.

Тепер спробуйте зайди на сторінку із списком студентів та переконайтесь, що дія Відвідування студента працює коректно та відображає лише даного студента у журналі відвідування. Також перевірте чи наша закладка Відвідування далі коректно працює. Адже ми заділи кілька компонент, на які дана сторінка залежить, тому важливо перевірити чи існуючий до цього моменту функціонал не поламався.

...

На даний момент ми повністю реалізували весь функціонал, що мав відношення до закладки Відвідування. Але поки Javascript коду написали зовсім небагато. Щоб краще зрозуміти основи використання мови Javascript та технології AJAX, продовжимо і реалізуємо ще кілька функцій та покращень у нашій аплікації. Кожна із функцій матиме невелику дозу залученого Javascript коду при реалізації.

Випадайка з групами

У нас вже давненько висить у верхньому правому кутку випадайка із міткою “Група:”. Ще у перших главах ми статично додали її і поки не поверталися до даного елементу.

The screenshot shows a web application titled 'Сервіс Обліку Студентів'. At the top, there is a navigation bar with tabs: 'Студенти' (Students), 'Відвідування' (Attendance), 'Групи' (Groups), and 'Контакт' (Contact). A red arrow points to a dropdown menu labeled 'Група:' which contains the option 'Усі Студенти' (All Students).

Випадайка з групами

До цього моменту ми вже навчилися усіх необхідних речей, щоб успішно оновити випадайку груп та зробити її динамічною. Для цього нам потрібно буде:

- зробити список груп динамічним і щоб брався із бази даних;
- написати Javascript код, який буде слідкувати за вибором групи у випадайці і запам'ятовуватиме його в куки; після цього даний код також буде редіректити на ту ж сторінку, але вже із переданими на сервер додатковими параметрами про групу;
- оновити в'юшки усіх сторінок із списками студентів, щоб працювали в контексті обраної групи.

Справжні групи у випадайці

Елемент із списком груп з'являється на кожній сторінці нашої аплікації і його вставляємо ми напряму у base.html шаблон. Це означає, що необхідні для даної випадайки дані прийдеться готовувати у коді кожної в'юшки.

Щоб уникнути дублювання коду, одразу вирішимо реалізувати генерацію даних груп у вигляді процесора контексту. Таким чином додавши логіку один раз на рівні процесора контексту, а ще краще в окремий модуль util.py, зможемо використовувати функції по роботі із групами у різних місцях нашої аплікації.

Але давайте спочатку оновимо наш шаблон base.html, щоб ми уявляли, які дані необхідно генерувати на сервері:

Динамізуємо список груп в base.html

```
1   <div class="col-xs-4" id="group-selector">
2     <strong>Група:</strong>
3     <select>
4       <option value="">Усі Студенти</option>
5       {% for group in GROUPS %}
6         <option value="{{ group.id }}"{% if group.selected %}selec\
7         ted="1"{% endif %}>{{ group.title }}{% if group.leader %}, {{ group.\
8         leader }}{% endif %}</option>
9         {% endfor %}
10        </select>
11    </div>
```

Як бачите, статичний список тегів option ми замінили динамічним циклом. Цикл пробігається по змінній-списку GROUPS. Дану змінну ми наступним кроком реалізуємо на рівні процесора контексту. Кожна опція отримує в якості значення - ID групи. Користувачу ж показуємо назву групи та ім'я старости, якщо, звичайно, староста призначений.

Також в кожній опції є перевірка на те, чи потрібно її вибирати як поточну.

Я спеціально позначаю змінні, що приходять із контексту процесора величими літерами, щоб розуміти походження тої чи іншої змінної. Загалом, немає значення як ви назовете свої змінні. Дане правило є швидше моєю особистою практикою для зручності.

Бракує лише самої змінної GROUPS. Для цього нам потрібно реалізувати ще один процесор контексту. Покладемо цього разу його в аплікацію students, а не в папку проекту, адже даний процесор занадто тісно зав'язаний на моделі, що йдуть із даною аплікацією.

Створюємо новий модуль для контексту процесора в корені аплікації students:

Створюємо context_processors.py в корені аплікації students

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students  
2 touch context_processors.py
```

Відкриваємо його у своєму редакторі та набиваємо дуже простою функцією:

Процесор контексту, який додає змінну GROUPS до усіх наших шаблонів

```
1 from .util import get_groups  
2  
3 def groups_processor(request):  
4     return {'GROUPS': get_groups(request)}
```

Тут немає жодної логіки. Лише визначення функції groups_processor, яка повертає словник із ключем 'GROUPS'. Значення для даного ключа формує функція get_groups, яку імпортуємо із модуля util.py.

Отже, давайте перейдемо до самої логіки генерації списку груп для нашого процесора контексту. Для цього відкриваємо наш існуючий модуль util.py в корені аплікації students і додаємо функцію get_groups:

Функція, яка повертає список студентів із бази

```
1 def get_groups(request):  
2     """Returns list of existing groups"""  
3     # deferred import of Group model to avoid cycled imports  
4     from .models import Group  
5  
6     # get currently selected group  
7     cur_group = get_current_group(request)  
8  
9     groups = []  
10    for group in Group.objects.all().order_by('title'):  
11        groups.append({  
12            'id': group.id,
```

```
13         'title': group.title,
14         'leader': group.leader and (u'%s %s' % (group.leader.fir\
15 st_name,
16             group.leader.last_name)) or None,
17         'selected': cur_group and cur_group.id == group.id and T\
18 rue or False
19     })
20
21     return groups
```

Давайте порядково розберемо дану функцію:

- 1ий рядок: функція під назвою `get_groups` приймає єдиний аргумент `request`; далі в коді побачите для чого він нам потрібен;
- 4ий: імпорт моделі групи робимо всередині функції; це так званий відкладений імпорт; модулі із моделями та в'юшками залежать на даний модуль `util.py` (тобто роблять із нього імпорти); в свою чергу, якщо ми також хочемо імпортувати з них об'екти для подальшої роботи, то можемо отримати ситуацію циклічного імпорту, коли аплікація не запускається; в таких випадках “відстрочка” для імпортів, як перенесення їх всередину модуля, функції, класу може добряче допомогти;
- 7ий: у випадайці нам потрібно буде знати поточно обрану групу, щоб вибрати її як активну в меню; також подібну інформацію нам потрібно буде у інших в'юшках при фільтруванні студентів обраної групи; саме тому ми вносимо визначення поточно обраної групи в окрему функцію під назвою `get_current_group`; вона приймає об'єкт запиту в якості аргументу, щоб отримати з нього дані про обрану користувачем групу; далі ми зайдемось реалізацією даної функції; вона також знаходитиметься в модулі `util.py`; якщо жодної групи не обрано (тобто показуємо усіх студентів), тоді дана функція поверне порожнє значення - `None`;
- 9ий: заготовочка порожнього списку для подальшого наповнення його існуючими групами;
- 10ий: пробігаємось по усіх групах з бази; думаю нічого нового для себе в даному рядку ви не знайшли;

- 11ий: додаємо новий елемент у список груп; кожен елемент складатиметься з ID групи, її назви, старости та індикатора чи група є поточно обраною;
- 14ий: знову використовуючи скорочений умовний синтаксис мови Python із логічними операторами, визначаємо чи поточна група має призначеного старосту, і якщо так - готуємо його повне ім'я у вигляді однієї об'єднаної стрічки;
- 17ий: ключ 'selected' позначає те, чи поточна група є обраною користувачем; робимо це порівнюючи ID поточно обраної групи із ID групи в поточній ітерації циклу;
- 20ий: наприкінці функції повертаємо список згенерованих груп.

На завершення роботи з Python кодом реалізуємо також функцію з визначення поточно обраної групи. Дану функцію також додаємо всередині модуля util.py:

Функція визначення поточно обраної групи

```
1 def get_current_group(request):
2     """Returns currently selected group or None"""
3
4     # we remember selected group in a cookie
5     pk = request.COOKIES.get('current_group')
6
7     if pk:
8         from .models import Group
9         try:
10             group = Group.objects.get(pk=int(pk))
11         except Group.DoesNotExist:
12             return None
13         else:
14             return group
15     else:
16         return None
```

Також давайте порядково її розберемо:

- 1ий рядок: функцію називаємо `get_current_group` та передаємо їй аргумент `request`; далі побачимо як даний запит ми використаємо;
- 5ий: як ми уже домовились, інформацію про обрану групу запам'ятовуватимемо в куки браузера; якщо ми просто передамо її в якості URL параметра, тоді при наступному запиті даний параметр вже не буде присутнім; це так звана властивість HTTP протоколу - `stateless` (без стану, немає зв'язку між попереднім і наступним запитами); саме тому, в даному випадку, дані, що знаходитимуться в куках браузера, допоможуть тримати дані на довший період часу, ніж один запит; в даному рядку коду ми дістаєму куки під назвою '`current_group`'; тут можете бачити для чого нам потрібен був об'єкт запиту - він містить атрибут `COOKIES`, в який Django веб-фреймворк поміщає усі куки вислані браузером на сервер; при кожному запиті браузер висилає усі куки даного домену і сторінки на сервер; далі, у наступних під-секціях, ми з вами навчимось встановлювати та маніпулювати браузерними куками через `Javascript` код;
- 7ий: якщо куки є, тоді пробуємо знайти обрану групу;
- 8ий: знову ж таки імпортуємо модель групи у “відстроченому” режимі;
- 9ий: огортаємо витягування групи з бази в `try/except` оператор, щоб уникнути помилок на інтерфейсі користувача;
- 10ий: пробуємо отримати обрану групу з бази; в `COOKIES` браузера ми зберігатимемо саме ID обраної групи;
- 12ий: якщо не знайдено групи за даною нам ID, тоді повертаємо порожнє значення;
- 14ий: інакше повертаємо знайдений об'єкт групи;
- 16ий: також повертаємо порожнє значення у випадку, якщо ми не знайшли куки під потрібним нам ключем.

Нам залишилось додати щойно реалізований процесор у список процесорів проекту. Для цього відкриваємо модуль `settings.py` в корені проекту `studentsdb` і додаємо елемент до списку `TEMPLATE_CONTEXT_PROCESSORS`:

settings.py

```
1 TEMPLATE_CONTEXT_PROCESSORS = \
2     global_settings.TEMPLATE_CONTEXT_PROCESSORS + (
3         "django.core.context_processors.request",
4         "studentsdb.context_processors.students_proc",
5         "students.context_processors.groups_processor",
6     )
```

На цьому все із серверною стороною для випадайки груп. Пробуйте перезавантажити вашу аплікацію і сторінку в браузері. Перегляньте список груп у випадайці. Вони мають співпадати із тими, що з'являються на закладці Групи.

Javascript код для випадайки

Наступним кроком додамо необхідний Javascript код, який встановлюватиме потрібний запис у браузерні куки та оновлятиме поточну сторінку. Перевантаження поточної сторінки потрібне для того, щоб відобразити її в контексті зміненої групи.

Javascript має прямий доступ до управління куками на поточному домені і сторінці. Але дефолтний інтерфейс роботи із куками є доволі незручний і багатослівний, тому скористаємося додатковою бібліотекою - плагіном для jQuery: [jQuery Cookie²⁸⁷](#).

З допомогою даного плагіну ми з легкістю зможемо створювати нову куку, а також оновляти та видялати існуючі. [Тут²⁸⁸](#) можете знайти коротку документацію з використання даної бібліотеки.

Підключимо дану бібліотеку, щоб мати доступ до необхідних функцій роботи з куками браузера. Для цього спочатку завантажимо останню стабільну версію даної бібліотеки і покладемо в папку “static/js/”:

²⁸⁷<http://plugins.jquery.com/cookie/>

²⁸⁸<https://github.com/carhartl/jquery-cookie#usage>

Завантажуємо Javascript файл бібліотеки jQuery Cookie і кладемо в папку static/js

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students/static\js
2 # це утиліта, яка дозволяє в командній стрічці
3 # завантажувати сторінки і файли з інтернету; ви можете
4 # використовувати будь-який зручний і знайомий вам
5 # інструмент для завантаження файлів, включно з браузером;
6 wget https://raw.githubusercontent.com/carhartl/jquery-cookie/v1.4.1\
7 /jquery.cookie.js
```

Потім відкриваємо base.html шаблон і додаємо ще один Javascript файл одразу після jQuery бібліотеки:

Додаємо jQuery Cookie плагін до нашого базового шаблону base.html

```
1 <!-- Javascripts Inclusion -->
2 <script src="http://ajax.googleapis.com/ajax/libs/jquery/2.1.1/jqu\
3 ery.min.js"></script>
4 <script src="{% static "js/jquery.cookie.js" %}"></script>
5 <script src="https://cdn.jsdelivr.net/bootstrap/3.3.0/js/bootstrap\
6 .min.js"></script>
```

Як бачите, ми вставили ще один script тег між jQuery та Twitter Bootstrap скриптами. При цьому скористались тегом static, щоб отримати динамічний та гнучкий статичний лінк.

Нарешті ми готові до написання Javascript коду. Отже, відкриваємо наш існуючий файл main.js і додаємо функцію initGroupSelector, яка ініціалізуватиме нашу випадайку з групами. Дану функцію також додаємо до “ready” події. Тобто наша функція буде запущена одразу після того як DOM буде готовий для нашого використання:

Функція обробки змін на випадайці груп

```
1 function initGroupSelector() {
2     // look up select element with groups and attach our even handler
3     // on field "change" event
4     $('#group-selector select').change(function(event){
5         // get value of currently selected group option
6         var group = $(this).val();
7
8         if (group) {
9             // set cookie with expiration date 1 year since now;
10            // cookie creation function takes period in days
11            $.cookie('current_group', group, {'path': '/', 'expires': 365}\
12        );
13        } else {
14            // otherwise we delete the cookie
15            $.removeCookie('current_group', {'path': '/'});
16        }
17
18        // and reload a page
19        location.reload(true);
20
21        return true;
22    });
23 }
24
25 $(document).ready(function(){
26     initJournal();
27     initGroupSelector();
28});
```

Оскільки є багато нових речей у даній функції, розберемо її детально:

- 1ий рядок: декларуємо функцію, яка чіплятиме наш код на подію change випадайки групи (з англ. change - зміна);

- 4ий: шукаємо на сторінці нашу випадайку через селектор “#group-selector select”, який шукатиме спочатку елемент з ID “group-selector”, а всередині нього тег під назвою select; до знайденого елемента чіпляємо обробник події change; це, як завжди, анонімна функція із першим аргументом - об'єктом події;
- 5ий: отримуємо значення обраного тегу option всередині випадайки з групами;
- 6ий: перевіряємо чи маємо обрану групу;
- 11ий: плагін jQuery Cookie розширив бібліотеку jQuery, а саме функцію “\$” (в даному контексті дана функція виступає швидше об'єктом), кількома додатковими функціями (cookie, removeCookie і ще кількома іншими); в даному рядку ми скористалися функцією cookie, яка створює куки згідно переданих їй аргументів: назва куки ‘current_group’, значення - ID обраної у випадайці груп, ‘path’ - шлях на сайті, до якого застосовується дана кука (у нашому випадку вона застосовується до усіх сторінок), ‘expires’ - кількість днів дії даної куки;
- 15ий: якщо жодної групи не було обрано, тоді видаляємо куку під назвою ‘current_group’; при видаленні обов'язково вказувати для яких шляхів видаляти дану куку;
- 19ий: ВОМ (об'єктна модель браузера) дає в нашему Javascript коді доступ до глобального об'єкта **location**²⁸⁹; він відповідає за адресу поточної сторінки; також з його допомогою можна оновлювати поточну сторінку і переходити на іншу; в даній стрічці ми скористались його методом reload, щоб оновити поточну сторінку; переданий параметр true змушує метод reload робити справжній запит на сервер, навіть, якщо є попередньо закешований запит на цю ж сторінку;
- 21ий: повертаємо true, щоб подія change тегу select спрацювала і на графічному інтерфейсі користувач отримав оновлене значення у випадайці;
- 27ий: додаємо нову функцію у список функцій, які запускатимемо, коли DOM уже готовий.

Перевантажте у форс-релоад режимі вашу сторінку і спробуйте поклікати. Якщо з певних причин сторінка не оновляється після вибору тої чи іншої

²⁸⁹<http://javascript.ru/window-location>

групи, перевірте чи все зробили згідно вищезгаданих прикладів. Зокрема, перевірте чи плагін jQuery Cookie підключений коректно.

Перший тест: сторінка повинна оновлюватись після зміни групи. Другий тест: повинна встановлюватись і оновлюватись кука ‘current_group’ у вашому браузері. Куки можна перевіряти в девелоперських інструментах кожного окремого браузера. Якщо ж користуєтесь браузером Firefox і плагіном Firebug, тоді у вас уже є [закладка Cookies](#)²⁹⁰, де можете переглянути як із кожним вибором групи значення даної куки змінюється. Якщо ж обираєте “Усі Студенти” у меню групи, тоді дана кука повинна зникати із списку.

Функція встановлення куки в дії

Якщо все гаразд і обидва тести пройшли “На ура!”, ми готові до переходу в наступну під-секцію, де доробимо головну частину всієї затії: оновимо список студентів так, щоб він зважав на нашу куку ‘current_group’.

Оновляємо серверну сторону для груп

Повертаємось до нашої першої в’юшки - в’юшки із списком студентів.

Все, що необхідно зробити, це перевіряти чи в даний момент є обрана певна група, і якщо так - віддавати список студентів, які є членами даної групи. В протилежному випадку, працювати без змін - відображати список усіх студентів.

²⁹⁰<http://getfirebug.com/cookies>

Отже, відкриваємо модуль в'юшки зі списком студентів у нашему редакторі і робимо невеликі правки:

Беремо до уваги обрану групу у в'ющі із списком студентів, `students/views.py`

```
1 from ..util import paginate, get_current_group
2
3
4 def students_list(request):
5     # check if we need to show only one group of students
6     current_group = get_current_group(request)
7     if current_group:
8         students = Student.objects.filter(student_group=current_group)
9     else:
10         # otherwise show all students
11         students = Student.objects.all()
12
13
14     # try to order students list
15     order_by = request.GET.get('order_by', '')
16     if order_by in ('last_name', 'first_name', 'ticket'):
17         students = students.order_by(order_by)
18         if request.GET.get('reverse', '') == '1':
19             students = students.reverse()
20
21     # apply pagination, 3 students per page
22     context = paginate(students, 3, request, {},
23                       var_name='students')
24
25     return render(request, 'students/students_list.html', context)
```

Що ж змінилось у даній функції:

- 1ий рядок: ми додатково імпортуємо функцію `get_current_group` з `util.py` модуля, щоб скористатись нею далі для визначення обраної групи;

- 6ий: отримуємо поточно обрану групу;
- 8ий: якщо група є вибраною, тоді з бази дістаємо список лише тих студентів, що належать до даної групи; для цього застосовуємо фільтр по полю зв'язку student_group;
- 12ий: інакше віддаємо повний список студентів.

Зауважте також, що дана функція було оновлена для використання функції paginate з модуля util.py. Це було одним із домашніх завдань з попередніх секцій книги.

Це все! Ще раз переконуємось як просто в Django робити досить кардинальні зміни. Поєднуючи це з кількома хвилинами роздумів, зазвичай, можете робити подібні трюки в лічені хвилини.

Спробуйте перевантажити вашу головну сторінку аплікації і вибирати групи. Потім вибрати “Усі Студенти”. Якщо кількість студентів змінюється, тоді швидше за все ваш код повністю готовий і працює коректно. Вітаю!

У випадку, якщо ви переписали ваші в'юшки на класи, тоді для застосування подібних змін, ваш код повинен йти в метод класу під назвою get_queryset. Логіка ж залишається такою самою.

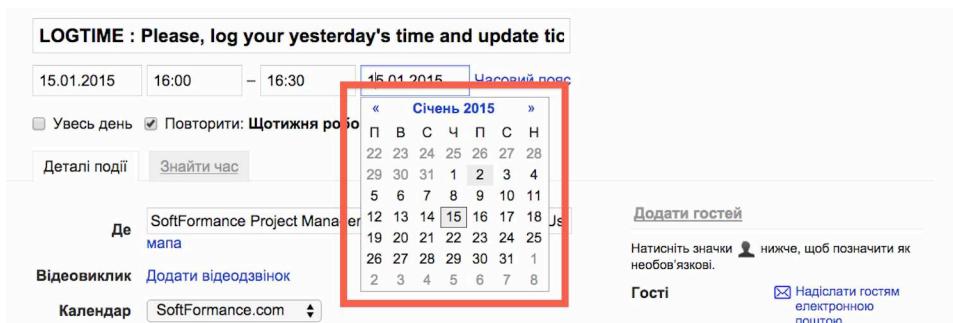
І що найцікавіше, наш фільтр по групах працює незалежно від того, на яку сторінку ви переходите і які дії робите на сайті. Аж допоки не оберете іншу групу, або опцію “Усі Студенти”, або мине цілий рік від дня останньої зміни групи, або ви скористаєтесь інших браузером або комп’ютером. Це все завдяки використанню Cookies в браузері.

На домашнє завдання залишаю вам подібним чином оновити логіку в'юшок груп, відвідування та іспитів, щоб вони брали до уваги обрану групу і працювали в її контексті. Так викладачу буде значно зручніше працювати із студентами саме його групи.

Віджет календаря для поля дати

Перед переходом до складнішого завдання з великою кількістю Javascript коду та застосування AJAX технології, зробимо ще одне невеличке покращення до наших форм. Приєднаємо на кожну форму до кожного поля дати віджет календаря. Адже значно зручніше кількома кліками мишко встановити потрібну дату, ніж на клавіатурі пробувати вгадати формат необхідної дати.

Віджети календаря вже стали повсякденною річчю при користуванні веб-формами. Щоб не відставати від моди і зробити життя наших користувачів простішим та продуктивнішим, також оновимо наші форми.



Приклад віджету календаря на формі, взятий із Google календаря

Звичайно, писати з нуля весь HTML, CSS та Javascript код ми не будемо, адже це доволі нетривіальне завдання, хоч і виглядає доволі просто на перший погляд. Натомість скористаємося готовим для нас Twitter Bootstrap плагіном [Bootstrap 3 Datepicker](#)²⁹¹.

Підключаємо плагін

Для початку підготуємо і підключимо усі необхідні ресурси, щоб можна було далі користуватись даним плагіном. Для цього скористаємося [інсталяційними інструкціями](#)²⁹², які можна знайти в репозиторії коду даного проекту.

²⁹¹<http://eonasdan.github.io/bootstrap-datetimepicker/>

²⁹²<https://github.com/Eonasdan/bootstrap-datetimepicker/wiki/Installation#manual>

jQuery у нас уже є на сторінці. Тому наступним кроком маємо підключити бібліотеку [moment.js²⁹³](#). Ця бібліотека необхідна для нашого плагіна, щоб працювати із різноманітними форматами дат.

Підключаємо її із існуючого CDN ресурсу: [cdnjs.com²⁹⁴](#). Гуглівський CDN містить лише обмежену кількість найбільш популярних бібліотек, в той час як cdnjs.com містить більшість необхідних бібліотек та фреймворків. Після швидкого пошуку знаходимо необхідний файл із Javascript кодом [Moment бібліотеки²⁹⁵](#).

Підключимо даний файл у base.html шаблоні одразу після jQuery скрипта. Порядок важливий:

Підключаємо бібліотеку moment.js

```
1 <!-- Javascripts Inclusion -->
2 <script src="http://ajax.googleapis.com/ajax/libs/jquery/2.1.1/jqu\
3 ery.min.js"></script>
4 <script src="http://cdnjs.cloudflare.com/ajax/libs/moment.js/2.9.0\
5 /moment.min.js"></script>
6 <script src="{% static "js/jquery.cookie.js" %}"></script>
7 <script src="https://cdn.jsdelivr.net/bootstrap/3.3.0/js/bootstrap\
8 .min.js"></script>
```

Наступним кроком підключимо Javascript файл самого плагіна. Його також можна знайти на [cdnjs.com²⁹⁶](#). Даний скрипт повинен йти обов'язково після Twitter Bootstrap скрипта:

²⁹³ <http://momentjs.com/docs/>

²⁹⁴ <https://cdnjs.com/>

²⁹⁵ <http://cdnjs.cloudflare.com/ajax/libs/moment.js/2.9.0/moment.min.js>

²⁹⁶ <http://cdnjs.cloudflare.com/ajax/libs/bootstrap-datetimepicker/4.0.0/js/bootstrap-datetimepicker.min.js>

Підключаємо файл Datepicker плагіна

```
1  <!-- Javascripts Inclusion -->
2  <script src="http://ajax.googleapis.com/ajax/libs/jquery/2.1.1/jqu\
3 ery.min.js"></script>
4  <script src="http://cdnjs.cloudflare.com/ajax/libs/moment.js/2.9.0\
5 /moment.min.js"></script>
6  <script src="{% static "js/jquery.cookie.js" %}"></script>
7  <script src="https://cdn.jsdelivr.net/bootstrap/3.3.0/js/bootstrap\
8 .min.js"></script>
9  <script src="http://cdnjs.cloudflare.com/ajax/libs/bootstrap-datet\
10 imepicker/4.0.0/js/bootstrap-datetimepicker.min.js"></script>
11 <script src="{% static "js/main.js" %}"></script>
```

До речі, jQuery Cookie плагін також можна знайти на CDN cdnjs.com. У попередніх секціях ми спеціально завантажили його локально, щоб по-практикуватись із використанням зовнішніх Javascript бібліотек в якості локально завантажених скриптів. Тепер, знаючи, що більшість популярних бібліотек можна знайти на cdnjs.com, можете переключитись на використання плагіну jQuery Cookie із зовнішнього ресурсу.

На завершення інсталяції необхідних ресурсів для віджета календаря підключимо файл із стилями:

Підключаємо Bootstrap Datepicker файл із стилями

```
1 <link rel="stylesheet"
2     href="https://cdn.jsdelivr.net/bootstrap/3.3.0/css/bootstrap\
3 .min.css">
4 <link rel="stylesheet"
5     href="http://cdnjs.cloudflare.com/ajax/libs/bootstrap-datetimepi\
6 picker/4.0.0/css/bootstrap-datetimepicker.min.css">
7 <link rel="stylesheet"
8     href="{{ PORTAL_URL }}{% static "css/main.css" %}">
```

Зauważте, що ми вклали даний файл після стилів бібліотеки Twitter Bootstrap та після наших кастомних стилів. Цей порядок важливий і дозволить, при потребі, кастомізувати кінцеві стилі власними правилами.

Перевантажте сторінку у браузері і переконайтесь, що статичні ресурси присутні на сторінці та дійсно вказують на коректні URL адреси із ресурсами.

Усе. Ми готові йти далі і додати віджет календаря до форми редагування студента.

Форма редагування студента

На сторінці редагування студента можемо знайти наступне поле “Дата народження”:

Форма редагування студента

Дане поле реалізоване наступним HTML кодом всередині тегу `<form>`:

HTML код поля дня народження

```

1 <input class="dateinput form-control" id="id_birthday"
2     name="birthday" required="required" type="text"
3     value="1986-04-11">

```

Це поле, якщо пам'ятаєте, для нас згенерувала аплікація Django Crispy Forms.

Наше завдання полягає у тому, щоб до даного поля “причепити” віджет календаря і правильно його налаштувати. Можемо скористатись ID або класом даного елемента, щоб під'єднати наш Javascript код.

Якщо б нам було потрібно додати календар лише до даного поля, тоді ідеально підійшов би атрибут `id`. Проте, нам потрібно мати даний віджет календаря на усіх полях дати та часу. В таких випадках краще залежати на атрибут класу (`class`), який буде спільним для усіх полів дати. Таким чином, написавши один раз Javascript код, зможемо заставити працювати його одразу для більшості необхідних полів.

Атрибут клас особливо корисний, якщо на усіх полях дат ми приєднуватимемо календарик з однаковими налаштуваннями. В протилежному випадку, для кожного поля потрібно буде окремо приєднувати та налаштовувати віджет. Відповідно, в такому випадку, краще прив'язуватись до ID елементів.

Ми зупинимось на класі. Для цього відкриємо наш власний Javascript файл main.js в редакторі та додамо необхідний мінімум коду, щоб приєднати календар до поля “Дата народження”:

```
1 ...
2
3 function initDateFields() {
4     $('#input.dateinput').datetimepicker({
5         'format': 'YYYY-MM-DD'
6     });
7 }
8
9 $(document).ready(function(){
10     initJournal();
11     initGroupSelector();
12     initDateFields();
13});
```

Кількість Javascript коду зовсім невелика. Тим не менше, давайте глянемо детальніше на нього:

- Зій рядок: декларуємо функцію initDateFields, яка займатиметься динамізацією усіх полів дати та часу;
- 4ий: шукаємо наше поле на сторінці і приєднуємо віджет календаря; під’єднаний плагін Datepicker розширив jQuery ще однією функцією: datetimepicker; тому кожен елемент сторінки, який ми отримуємо через jQuery функцію “\$”, матиме даний метод datetimepicker; він “чіпляє” календар; даному методу передаємо набір налаштувань²⁹⁷ у вигляді Javascript об’єкта (в мові Python цей тип даних називаємо словником);
- 5ий: задаємо формат дати для віджета календаря; він повинен співпадати із форматом, який ми раніше обрали на серверній стороні (або за нас це зробив клас роботи із Django формою), інакше календар ламатиме його і користувач не зможе відправити форму на сервер; тут²⁹⁸ можна пе-

²⁹⁷ <http://eonasdan.github.io/bootstrap-datetimepicker/Options>

²⁹⁸ <http://momentjs.com/docs/#/displaying/format/>

реглянути усі доступні опції щодо конфігурації формату дати в нашому плагіні;

- 12ий: не забуваємо додати виклик нашої нової функції всередину обробника завантаження сторінки.

Спробуйте форс-релоаднути форму редагування студента і клацніть по полю “Дата народження”. Приблизно такий результат має бути у вашому браузері:

April 1986						
Su	Mo	Tu	We	Th	Fr	Sa
30	31	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	1	2	3

Віджет календаря на полі з датою

Спробуйте покористуватись даним календарем та зберегти зміни на сервер. В процесі можете зауважити одну незручність. Після вибору дати на віджеті поле дати залишається сфокусованим. Тому, щоб повторно отримати віджет календаря, потрібно спеціально зняти фокус з даного поля (клацнувши на будь-якому іншому місці сторінки аніж полю), а тоді повторно клікнути всередині поля дати.

Щоб уникнути даної незручності, потрібно після вибору дати на віджеті запустити Javascript код, який зніматиме фокус з поточного поля. Це можна зробити через додаткові події, які запускає плагін календаря. Ми прив’яжемось до події `dp.hide299`:

²⁹⁹<http://eonasdan.github.io/bootstrap-datetimepicker/Events/#dphide>

Знімаємо фокус після того, як дата обрана в календарі і віджет захованний

```
1 function initDateFields() {  
2     $('input.dateinput').datetimepicker({  
3         'format': 'YYYY-MM-DD'  
4     }).on('dp.hide', function(event){  
5         $(this).blur();  
6     });  
7 }
```

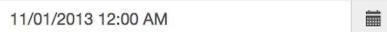
До нашої функції ми додали 2 нових рядка:

- 4ий рядок: кожен метод в jQuery повертає об'єкт над яким він працює; таким чином datetimepicker метод поверне нам поточне поле дати; ми уже “чіпляли” обробники подій через назву події (напр. [об'єкт].click(функція)), але це не єдиний метод; є також можливість використати метод “on”, якому передати ім'я події, на яку приєднуємо наш обробник; у даному випадку ми змушені скористатись методом “on” тому, що назва події містить крапку, яка інтерпретуватиметься як доступ до атрибута об'єкту; тому передаємо назvu події у вигляді стрічки з використанням методу “on”; другим аргументом даного методу передаємо функцію-обробник події; подія ‘dp.hide’ запускається плагіном календаря, коли календар заховано із сторінки;
- 5ий: до об'єкта, з яким відбувалась подія, всередині обробника події, маємо доступ через ключове слово “this”; завдяки обгортанню його в jQuery об'єкт, маємо доступ до потрібних нам методів; у нашему випадку це blur; даний метод знімає фокус із поля форми.

Перевантажте сторінку і спробуйте повторно відкрити календар. Користуватись даним віджетом тепер стало значно зручніше. Це зайняло лише 2 рядки Javascript коду.

На даному етапі завершуємо роботу із календарями. Проте є ще маса шляхів, щоб ще покращити даний віджет на усіх наших формах. Залишаю кілька завдань на самостійне опрацювання.

Щоб користувач бачив, що від даного поля варто очікувати календарик, можна поставити додаткову кнопку справа від поля. Вона слугуватиме індикатором додаткового функціоналу. Для цього прийдеться розібратись як кастомізнути HTML код поля дати, що є згенероване з допомогою автоматичної форми. Реалізацію HTML/CSS коду можна взяти із Twitter Bootstrap. Ось як може виглядати дана кнопка:



Поле дати із кнопкою-індикатором

Також на домашнє опрацювання можна перекласти даний календарик на українську мову. Плагін дає можливість локалізації. Для цього треба буде підключити ще один файл JavaScript коду із перекладами і налаштувати Datepicker плагін. Шукайте деталі в документації плагіна.

І на завершення домашніх завдань даної секції, пропоную також додати календар до форм додавання і редагування Студента та Іспитів.

Форма редагування студента в режимі AJAX

Наступним завданням буде реалізація форми редагування студента без потреби перевантаження сторінки. Замість окремої сторінки із формою користувач, при кліку по лінку Редагувати, динамічно отримуватимемо окреме вікно всередині існуючої сторінки. В даному вікні можна буде працювати із формою

редагування студента без перевантаження сторінки. Лише успішне збереження студента закриватиме дане вікно і перевантажуватиме сторінку списку студентів, щоб показати оновленого студента.

Сервіс Обліку Студентів

База Студентів

Студенти Відвідування

Додати Студента

#	Фото
1	
2	
3	

« 1 2 3 4 »

© 2014 Сервіс Обліку Студентів

Редагувати Студента

Ім'я* Віталій

Прізвище* Подоба

По-батькові Іванович

Дата народження* 1984-06-17

Фото Нараїз: ./podoba3.jpg Очистити
Змінити: Choose File No file chosen

Білет* 3851

Група* Demo Group 1 (Іван Лісовий)

Дії

- Редагувати
- Відвідування
- Видалити

Зберегти Скасувати

Форма редагування студента в режимі AJAX

AJAX і Javascript динамізацію функціоналу завжди можна реалізувати масою різноманітних підходів. Немає певних правил чи затверджених обмежень, які можуть допомогти початківцю створювати динамічні веб-інтерфейси не наламавши дров.

Наприклад, у нашому випадку ми можемо реалізувати існуючу форму редактування як мінімум наступними трьома способами:

- оновити серверну сторону, яка б віддавала два різних варіанта HTML коду сторінки із формою редактування взалежності від типу запиту: звичайний чи AJAX; відповідно, в такому випадку основна логіка по обслуговуванню різних запитів лягає на сервер;
- написати Javascript код, який перевикористає вже існуючий код на сервері і виконає необхідні модифікації із HTML кодом, щоб підпасувати його до потреб AJAX сторінок і функціоналу;
- зробити мікс двох вищеперелічених варіантів: частину логіку помістити на сервер (у нашему випадку - це Python), а частину на клієнт (Javascript).

Рекомендую уникати третього варіанту, оскільки він призводить до складнішої підтримки вашої аплікації. Адже при будь-яких оновленнях функціоналу проекту вам прийдеться, в більшості випадків, оновляти як серверну так і клієнтську частину.

У нашому випадку ми скористаємось другим підходом і всю логіку по реалізації форми редагування закладемо у Javascript код. Це не завжди буває можливим і деколи приходиться робити також зміни і на сервері. Все залежить від складності динамічного функціоналу та того, як реалізована серверна сторона.

Для форми редагування студента обійтися одним Javascript кодом без змін логіки в'юшок на сервері. Цей підхід також дозволить нам більше повправлятись із мовою Javascript та AJAX підходом. Що і є ціллю даної глави книги.

В наш час є надзвичайно розповсюджені динамічні веб-аплікації, які містять 50% коду і логіки на стороні клієнта. Соцмережі активно використовують чати, динамічні стіни та нотифікації, які є частиною “живого” інтерфейсу. Для подібного роду проектів та завдань розробники реалізували масу Javascript бібліотек та фреймворків: Backbone, AngularJS, Ember і т.п. Усі вони дозволяють ефективніше розробляти динамічні аплікації в режимі ООП і не губитись у десятках тисяч Javascript коду. Саме тому останнім часом фронт-енд розробка стала окремим напрямком і хороши HTML/CSS/Javascript спеціалісти є дуже затребувані на ринку. Як розробляти подібні аплікації є окремим великим предметом і не охоплений матеріалами даної книги.

Ось список завдань, які нам потрібно зробити для реалізації форми редагування студента в режимі AJAX:

- підключити плагін для роботи форм в режимі AJAX;
- вирішити, що використовуватимемо в якості Javascript вікон для нашої форми;
- додати заготовку вікна в HTML шаблони;
- ну і, звичайно, написати Javascript код, який відкриватиме вікно із формою і динамізуватиме її обробку.

Інсталюємо додаткові інструменти

Почнемо з інсталяції і налаштування додаткових інструментів необхідних для реалізації AJAX форми.

Оскільки ми вже використовуємо цілий ряд Javascript бібліотек та фреймворків, я рекомендую, першим ділом глянути чи дають вони нам частину необхідних інструментів.

Наприклад, Twitter Bootstrap фреймворк надає так звані [модальні вікна](#)³⁰⁰. Можемо ними скористуватись для реалізації вікна із формою редагування. Модальне вікно дасть нам необхідний мінімум функціоналу:

- прив'яжеться до кліку по кнопці чи лінку і покаже саме вікно;
- дозволить показати наш власний контент всередині вікна;
- закриє вікно у необхідний момент.

Всередині вікна ми надаватимемо користувачу можливість редагувати студента. Основною ціллю нашої затії є зробити роботу із формами для користувача максимально швидкою та продуктивною. Відповідно було б добре, якщо і пост форми, і валідація, і повідомлення про успішну обробку форми - усі ці речі відбувались без перевантаження сторінки.

Звичайно, ми можемо написати Javascript код, який ловитиме клік по кнопці, збиратиме дані з форми, відправлятиме їх на сервер, отримуватиме відповідь, показуватиме підсвічені проблемні поля, очищатиме поля після успішної обробки, і т.д. Але більшість речей у даному списку є занадто поширеними при розробці веб-аплікацій, щоб наново винаходити велосипед.

До нас уже створили масу бібліотек, які можуть значно полегшити реалізацію AJAX форми. Сама бібліотека jQuery не надає подібного функціоналу, але існує досить популярний плагін, що дозволить нам перевести будь-яку веб-форму на AJAX режим: [jQuery Form Plugin](#)³⁰¹.

Підключимо його в шаблоні base.html одразу після бібліотеки jQuery:

³⁰⁰<http://getbootstrap.com/javascript/#modals>

³⁰¹<http://malsup.com/jquery/form/>

Підключаємо jQuery Form Plugin, base.html шаблон

```
1  <!-- Javascripts Inclusion -->
2  <script src="http://ajax.googleapis.com/ajax/libs/jquery/2.1.1/jqu\ 
3 ery.min.js"></script>
4  <script src="http://cdnjs.cloudflare.com/ajax/libs/jquery.form/3.5\ 
5 1/jquery.form.min.js"></script>
```

Як бачите, jquery.form.min.js скрипт ми також знайшли на cdnjs.com ресурсі. Тому немає змісту використовувати наперед завантажену локальну копію скрипта, який гарантовано буде повільніше працювати на вашому продакшин сервері.

Маємо інструменти для реалізації модальних вікон та AJAX форм. Все решта зробимо самі. Рухаємось до шаблонів!

Оновлюємо шаблони

В даній під-секції нам потрібно зробити дві речі, щоб перейти до написання необхідного Javascript коду:

- в списку студентів є три лінка, які вказують на форму редагування студента: прізвище студента, його ім'я та лінк Редагування в меню дій; щоб одним селектором в Javascript коді приєднати той самий обробник кліка мишкої, додамо до кожного із даних лінків однакові класи; це трохи упростить нам життя при написанні Javascript коду;
- для того, щоб на сторінці з'являлось модальне вікно, потрібно підготувати заготовку, яка задаватиме структуру вікна; а вже у нього, при кожному кліку по лінку редагування, ми вставлятимемо різний контент.

Щоб виконати перше завдання, відкриємо шаблон, що відповідає за список студентів і додамо новий клас до лінків редагування студента:

Оновлюємо лінки редагування студента, students_list.html

```
1 ...
2
3     <td><a href="{% url "students_edit" student.id %}" class="stud\ent-edit-form-link">{{ student.last_name }}</td>
4     <td><a href="{% url "students_edit" student.id %}" class="stud\ent-edit-form-link">{{ student.first_name }}</td>
5
6 ...
7
8 ...
9
10        <ul class="dropdown-menu" role="menu">
11            <li><a href="{% url "students_edit" student.id %}" class="student-edit-form-link">Редагувати</a></li>
12            <li><a href="{% url "journal" student.id %}">Відвідуванн\я</a></li>
13            <li><a href="{% url "students_delete" student.id %}">Вид\алити</a></li>
14        </ul>
15 ...
16 ...
17 ...
18 ...
19 ...
```

Вище наведені два окремих куска HTML коду із шаблону students_list.html. Перший - це дві комірки рядка із студентом, де наведено Прізвище та Ім'я студента. До кожного із лінків ми додали клас “student-edit-form-link”. Подібним чином у другому куску HTML коду ми додали цей клас до лінка Редагувати всередині меню Дій. Далі ми зможемо приєднати необхідні обробники подій до усіх трьох лінків завдяки класу “student-edit-form-link”.

Наступним кроком підготуємо необхідний мінімум HTML коду для роботи із модальними вікнами. Всю документацію щодо роботи з модальними вікнами ви можете знайти на сторінці [Twitter Bootstrap³⁰²](http://getbootstrap.com/javascript/#modals). Звідти ми просто візьмемо перший приклад коду, трохи змінимо і вставимо в base.html шаблон одразу перед Javascript файлами:

³⁰²<http://getbootstrap.com/javascript/#modals>

Заготовка для модального вікна

```
1 ...
2
3 <!-- Modal Boilerplate -->
4 <div class="modal fade" id="myModal" tabindex="-1" role="dialog"
5     aria-labelledby="myModalLabel" aria-hidden="true">
6     <div class="modal-dialog">
7         <div class="modal-content">
8             <div class="modal-header">
9                 <button type="button" class="close" data-dismiss="modal"
10                    aria-label="Close">
11                     <span aria-hidden="true">&times;</span></button>
12                     <h4 class="modal-title" id="myModalLabel"><!-- --></h4>
13             </div>
14             <div class="modal-body"><!-- --></div>
15             <div class="modal-footer"><!-- --></div>
16         </div>
17     </div>
18 </div>
19
20 <!-- Javascripts Inclusion -->
21
22 ...
```

В скопійованому прикладі коду для модального вікна ми забрали увесь вміст і залишили шапку, футер та тіло вікна порожніми. Давайте детальніше розглянемо HTML код:

- 4ий рядок: контейнер для усіх елементів в модальному вікні; клас “modal” надає стилі вікна, а клас “fade” початково ховає даний елемент на сторінці; встановлюємо id атрибут, щоб потім по ньому прив’язати наш Javascript код; атрибути role, aria-labelledby та aria-hidden додаємо до нашого тегу, щоб Twitter Bootstrap Javascript код ігнорував елементи вікна і не застосовував ніяких особливих дій; завдяки даним атрибутам ми вказуємо, що Twitter Bootstrap має справу з модальним вікном;

- бий: цей та наступний вкладений теги необхідні для правильного функціонування вікна та застовування необхідних стилей самою бібліотекою Twitter Bootstrap; вони є обов'язковими;
- 8ий: тег div з класом “modal-header” містить верхню частину модального вікна і містить заголовок та кнопку, щоб закрити вікно; атрибут data-dismiss повідомляє Twitter Bootstrap фреймворк про те, що дана кнопка повинна закривати модальне вікно;
- 11ий: тег span міститиме іконку, щоб при кліку по ній закрити вікно;
- 12ий: тут буде заголовок модального вікна, який наповнюватимемо текстом при кожному завантаженні і відображені вікна; а поки залишаємо його порожнім;
- 14ий: заготовка-контейнер для тіла модального вікна; також порожня;
- 15ий: заготовка для футера вікна; вона нам не пригодиться для форми редагування студента, але залишаємо його для повної структури модального вікна.

Якщо даний код вставите в шаблон base.html на кореневому рівні всередині тега body, то його не побачите у браузері. Щоб скористуватись даною заготовкою HTML коду, переходимо до найцікавішої частини - Javascript обробник кліків на лінках редагування студента.

Javascript код

Перед тим як перейдемо до написання самого коду, давайте спочатку розберемось, в чому полягає задача:

- прив'язати обробник кліків по лінках редагування студента;
- на кожен клік завантажувати форму редагування із сервера і показувати її в модальному вікні, яке відкриваємо одразу після завантаження форми;
- до форми редагування прив'язати Javascript обробник поста форми;
- на пості форми відправляти запит на сервер;
- отримати відповідь від сервера і відобразити або форму із помилками;
- або повідомлення про успішно збережені дані;

- а також при успішному збереженні оновляти поточну сторінку (для цього ми просто зробимо браузерне оновлення, а не AJAX; це трохи полегшить нам задачу).

Почнемо із заготовки функцій та прив'язки нашого обробника до кліків по лінках, які ми напередодні підготували в шаблоні `students_list.html`. Для цього знову відкриваємо наш вже добре знайомий файл `main.js` і додаємо наступний код:

Заготовка для модальних вікон

```
1 function initEditStudentPage() {
2     $('a.student-edit-form-link').click(function(event){
3         var modal = $('#myModal');
4         modal.modal('show');
5         return false;
6     });
7 }
8
9 $(document).ready(function(){
10     initJournal();
11     initGroupSelector();
12     initDateFields();
13     initEditStudentPage();
14 });
```

Даний код лише показуватиме порожнє модальне вікно при кожному кліку по лінках форми редагування студента:

- 1ий рядок: функція `initEditStudentPage` міститиме більшість коду по роботі із модальними вікнами для форм редагування студента;
- 2ий: шукаємо усі лінки на сторінці із класом ‘`student-edit-form-link`’ (це той клас, який ми раніше додали до потрібних посилань у списку студентів) та приєднуємо функцію-обробник події `click`;
- 3ий: всередині обробника шукаємо контейнер модального вікна;

- 4ий: та показуємо знайдене модальне вікно; Twitter Bootstrap фреймворк розширив jQuery бібліотеку функцією modal; нею ми користуватимемося, щоб створювати, конфігурувати та управляти модальними вікнами; у даному випадку ми викликали її з аргументом ‘show’, який просто дає команду показати вікно;
- 5ий: повертаємо false із функції, щоб браузер не перейшов по лінку, а просто його проігнорував; ми хочемо, щоб дані лінки тепер лише відкривали модальні вікна; повернення false із функції-обробника події повідомляє браузер ігнорувати подію і не застосовувати свого власного обробника;
- 13ий: запускаємо дану функцію лише після повного завантаження сторінки.

Спробуйте оновити сторінку і потестувати щойно доданий код. На кожен клік по імені, прізвищу чи меню Редагувати повинні отримувати порожнє модальне вікно.

Давайте тепер зробимо так, щоб наш обробник робив кориснішу річ: відображав форму редагування обраного студента. Для цього нам потрібно буде робити AJAX запит на сервер за формою і лише тоді, коли її отримали, відображати модальне вікно.

Ускладнюємо код нашого обробника події:

Робимо запит на сервер і відображаємо форму редагування в модальному вікні

```
1 function initEditStudentPage() {
2     $('a.student-edit-form-link').click(function(event){
3         var link = $(this);
4         $.ajax({
5             'url': link.attr('href'),
6             'dataType': 'html',
7             'type': 'get',
8             'success': function(data, status, xhr){
9                 // check if we got successfull response from the server
10                if (status != 'success') {
11                    alert('Помилка на сервері. Спробуйте будь-ласка пізніше.');
```

```

12         return false;
13     }
14
15     // update modal window with arrived content from the server
16     var modal = $('#myModal'),
17         html = $(data), form = html.find('#content-column form');
18     modal.find('.modal-title').html(html.find('#content-column h\
19 2')).text());
20     modal.find('.modal-body').html(form);
21
22     // setup and show modal window finally
23     modal.modal('show');
24 },
25 'error': function(){
26     alert('Помилка на сервері. Спробуйте будь-ласка пізніше.');
27     return false;
28 }
29 });
30
31 return false;
32 });
33 }

```

Код функції значно зріс. Основну частину його складає запит на сервер по форму редагування та відображення даної форми у модальному вікні. Тут варто знову ж таки пройтись по кожному новому рядку коду:

- Зій рядок: запам'ятуємо об'єкт лінка, по якому клікнули; ми ще неодноразово звертатимемось до нього;
- 4ий: робимо **AJAX**³⁰³ запит на сервер, щоб отримати HTML код форми для обраного студента;
- 5ий: передаємо адресу, на яку потрібно зробити запит; адресу беремо з атрибути лінка "href"; ми уже домовились, що використовуватимемо

³⁰³<http://jquery-docs.ru/ajax/>

HTML код сторінки із формою редагування; тому усі запити на сервер, що ми до цього моменту реалізували, будемо використовувати і для AJAX запитів; даний запит поверне нам повний HTML код сторінки, а вже з нього ми пізніше виберемо лише те, що потрібно;

- бий: атрибутом ‘`dataType`’ ми вказуємо на тип даних, який очікуємо із сервера; у нашому випадку це буде HTML код;
- 7ий: тип запиту GET, адже ми лише отримуємо дані із сервера і нічого не записуємо в базу;
- 8ий: функція-обробник успішного запиту на сервер; її аргументи ми вже обговорювали з вами раніше, коли реалізували закладку Відвідування;
- 10ий: першим ділом даний обробник перевіряє чи маємо справу з успішним завершенням запиту на сервері; якщо ні, тоді показуємо браузерне вікно із повідомлення про помилку та просимо спробувати ще раз трохи пізніше;
- 16ий: шукаємо контейнер модального вікна і присвоюємо його змінній `modal`;
- 17ий: зауважте як ми через кому визначаємо кілька змінних із одним ключовим словом `var`; тут ми огортаємо дані, що прийшли із сервера, в jQuery об’єкт з допомогою функції “`$`”; після цього ми зможемо виконувати багато різноманітних маніпуляцій над HTML кодом, що прийшов із сервера; зокрема, шукати елементи всередині даного HTML коду, як це робимо для того, щоб знайти форму всередині елемента з ID “`content-column`”; присвоюємо знайдену форму змінній `form`; пізніше ми її вставимо у поки порожнє модальне вікно;
- 18ий: отриманий HTML код із сервера містить усю сторінку із формою; але нам потрібно лише кілька елементів (в тому числі сама форма), щоб відобразити їх у модальному вікні; у цьому рядку коду ми шукаємо контейнер заголовка в модальному вікні і вставляємо у нього заголовок сторінки, який знаходимо в отриманому із сервера HTML коді з допомогою селектора: ‘`#content-column h2`’; знайшовши тег заголовку сторінки `h2`, ми витягуємо лише його текст і вставляємо його у модальне вікно (для цього використовуємо jQuery метод `text304`); для оновлення коду тегу ми використовуємо метод `html305` jQuery бібліотеки; даний метод

³⁰⁴<http://jquery-docs.ru/attributes/text/>

³⁰⁵<http://jquery-docs.ru/attributes/html/>

може отримувати як jQuery об'єкт, так і стрічку HTML коду; щоб знайти внутрішній елемент в отриманому HTML документі ми скористались методом `find306`;

- 20ий: подібним чином оновлюємо код тіла модального вікна, але цього разу передаємо методу `html` змінну `form`, що містить форму редагування студента; результатом даного виклику буде HTML код форми всередині тіла модального вікна;
- 23ій: коли модальне поле набите необхідним контентом (заголовок і тіло вікна), можемо нарешті показати його користувачу; для цього використовуємо той самий код, що був у нас ще на етапі заготовки функції;
- 25ий: передаємо функцію-обробник помилкового стану на сервері; якщо помилка таки трапиться, тоді ми просто показуємо користувачу повідомлення про помилку і просимо спробувати ще раз пізніше.

Знову спробуйте оновити сторінку і поекспериментувати із новим модальним вікном. Цього разу воно вже буде заповненим. Спробуйте запостити форму. Побачите, що браузер оновить оригінальну сторінку і відобразить статусне повідомлення із результатом ваших дій.

Справа в тому, що ми ще не приєднали жодних Javascript обробників поста форми і вона, звісно, працює в стандартному режимі - постить дані на сервер звичайним браузерним запитом, а не AJAX-овим.

Наступним кроком напишемо Javascript код, який зробить можливим пост форми без перезавантаження сторінки. У цьому нам допоможе плагін `jQuery Form`.

Код щодо ініціалізації AJAX форми помістимо в окрему функцію. Нам потрібно буде активувати AJAX функції форми кожного разу після поста форми, якщо матимемо помилки при валідації. Огортання у функцію частини коду допоможе нам уникнути дублювання і з легкістю застосовувати необхідну ініціалізацію у будь-який момент одним викликом функції.

³⁰⁶<http://jquery-docs.ru/traversing/find/>

Звичайно така оптимізація коду не приходить до програміста з першої спроби. Якщо б я вперше робив подібного роду функціонал форми, я б помістив спочатку код у обробник “success” AJAX запиту, а вже потім, зрозумівши, що знову треба оновляти форму, переніс в окрему функцію. З часом такі фішки прийдуть і до вас. На кожен функціонал, який реалізовуватимете більше, ніж один раз, у вас з’являтимуться свої власні підходи, техніки та заготовки. Саме так, зазвичай, і зароджуються бібліотеки та фреймворки.

Після того, як всередині функції-обробника ‘success’ AJAX запиту вставляємо форму у модальне вікно, одразу викличемо функцію ініціалізації форми:

Прикріплюємо AJAX поведінку до нашої форми

```
1 // update modal window with arrived content from the server
2 var modal = $('#myModal'),
3     html = $(data), form = html.find('#content-column form');
4 modal.find('.modal-title').html(html.find('#content-column h\
5 2')).text());
6     modal.find('.modal-body').html(form);
7
8     // init our edit form
9     initEditStudentForm(form, modal);
10
11    // setup and show modal window finally
12    modal.modal('show');
```

В 9му рядку ми додали виклик функції `initEditStudentForm`, яку реалізуємо у наступному кроці. Данна функція приймає об'єкт форми та модального вікна. Основним її завданням буде зробити так, щоб форма редагування працювала без перевантажень сторінки браузера і обробляла як помилкові так і успішні пости. Ось в деталях список речей, які дана функція має реалізувати:

- при завантаженні форми у модальне вікно: “навішати” віджет календаря; форма динамічно вставляється Javascript кодом, а тому браузер нам

тут не допоможе і не викличе потрібні функції як це робить при завантаженні сторінки; таким чином, ми повинні самі викликати необхідні функції після того як додамо форму у вікно;

- при кліку по кнопці Скасувати: закрити модальне вікно;
 - при кліку по кнопці Зберегти: зробити запит на сервер із даними з форми, отримати відповідь і відобразити у модальному вікні; якщо успішно, тоді також оновити поточну сторінку.

Де-небудь на кореневому рівні файлу main.js додаєте наступну функцію:

Функція ініціалізації АЈАХ форми

```
1 function initEditStudentForm(form, modal) {
2     // attach datepicker
3     initDateFields();
4
5     // close modal window on Cancel button click
6     form.find('input[name="cancel_button"]').click(function(event){
7         modal.modal('hide');
8         return false;
9     });
10
11    // make form work in AJAX mode
12    form.ajaxForm({
13        'dataType': 'html',
14        'error': function(){
15            alert('Помилка на сервері. Спробуйте будь-ласка пізніше.');
16            return false;
17        },
18        'success': function(data, status, xhr) {
19            var html = $(data), newform = html.find('#content-column form'\
20);
21
22            // copy alert to modal window
23            modal.find('.modal-body').html(html.find('.alert'));
24
```

```
25      // copy form to modal if we found it in server response
26      if (newform.length > 0) {
27          modal.find('.modal-body').append(newform);
28
29          // initialize form fields and buttons
30          initEditStudentForm(newform, modal);
31      } else {
32          // if no form, it means success and we need to reload page
33          // to get updated students list;
34          // reload after 2 seconds, so that user can read
35          // success message
36          setTimeout(function(){location.reload(true);}, 500);
37      }
38  }
39 });
40 }
```

Як бачите, коду доволі багато, тому порядково розберемо нашу нову функцію:

- 1ий: декларуємо функцію initEditStudentForm, яка приймає аргументами об'єкт форми та модального вікна; далі в коді функції ми ними активно користуватимемось;
- 3ий: викликаємо нашу функцію initDateFields, яка приєднає віджет календаря до усіх знайдених на сторінці полів дат; нашу форму ми додали щойно на сторінку у модальне вікно (яке є поки прихованим), а функція initDateFields була запущена браузером одразу після завантаження сторінки; тому будь-які нові елементи на сторінці маємо ініціалізувати самостійно власним Javascript кодом;
- 6ий: працюємо із кнопкою Скасувати; знаходимо її на сторінці та чіпляємо обробник події click;
- 7ий: функція просто ховатиме модальне вікно; для цього використовуємо переданий об'єкт - контейнер модального вікна і на ньому викликаємо метод modal, якому цього разу передаємо аргумент "hide", що означає "сховати"; зауважте, що у Javascript функції маємо доступ до

змінних визначених поза даною функцією; завдяки цьому маємо доступ до змінної modal переданої аргументом у функцію initEditStudentForm;

- 8ий: повернення false із функції вказує браузеру відмінити дефолтну поведінку при кліку по кнопці;
- 12ий: найцікавіше місце у нашій функції; тут ми використовуємо плагін [jQuery Form³⁰⁷](#); він розширив бібліотеку jQuery функцією ajaxForm; дана функція запускається в якості метода над об'єктом форми і виконує для нас усю важку роботу; вона подібна до методу аjax по функціоналу, але також збирає за нас усю інформацію з форми; також ajaxForm дає нам інтерфейс для роботи із відповіддю, що приходить від сервера подібним чином як це робить метод аjax; щоб налаштувати нашу форму правильним чином, передаємо методу ajaxForm ряд параметрів;
- 13ий: так само як і в аjax методі, ключ 'dataType' означає тип даних, які прийдуть із сервера; у нашему випадку це HTML код;
- 14ий: функція-обробник випадку, коли із сервера отримуємо помилку; так само як і у попередніх випадках, просто показуємо повідомлення про помилку у браузерному вікні; ви можете спробувати реалізувати кастомний обробник помилки, який показуватиме дане повідомлення зверху форми в модальному вікні у вигляді статусного повідомлення;
- 18ий: основна маса коду входить у функцію-обробник успішного запиту на сервер; даний обробник отримує аналогічні аргументи як і у випадку з функцією аjax: дані із сервера (у нашему випадку це HTML код форми), стан відповіді, об'єкт запиту; до речі, частина цієї функції є подібною до того, що ми робимо в 'success' обробнику функції аjax при запиті на сервер по саму форму при початковому завантаженні модального вікна;
- 19ий: перетворюємо HTML текст в jQuery об'єкт, щоб пізніше маніпулювати об'єктами в HTML документі, що прийшов із сервера; застосовуємо цей об'єкт у змінну html; також в даному рядку шукаємо форму у коді, що прийшов із сервера і зберігаємо її у змінну newform; ми спеціально вибираємо називу для цієї змінної, щоб відрізнялась і не конфліктувала із попередньо переданою змінною form (формою, що була напередодні запущеною із модального вікна);
- 23ій: тут ми перетираємо поточний вміст "div.modal-body" елементу в модальному вікні статусним рядком, що прийшов із сервера; ще раз

³⁰⁷<http://malsup.com/jquery/form/>

нагадаю, що це прийнята практика робити кілька викликів методів в jQuery в один рядок, тим самим зменшуючи кількість коду; в даному рядку ми в модальному вікні шукаємо елемент “modal-body” (метод find), потім у ньому перетираємо уесь вміст (метод html); ми завжди отримуватимемо статусне повідомлення після поста форми будь-то повідомлення про успіх чи невдачу (валідація помилок);

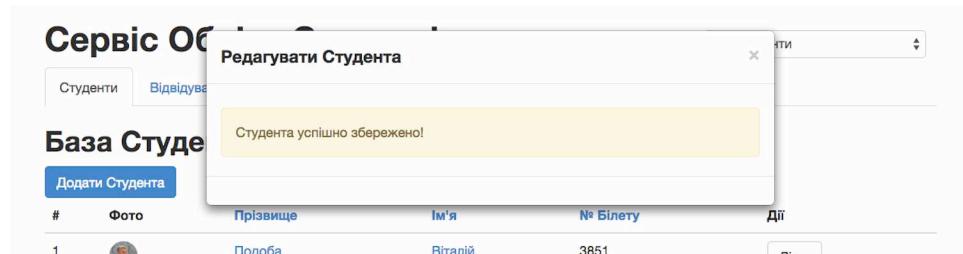
- 26ий: jQuery функції, які шукають елементи на сторінці по селектору завжди повертають не один елемент, а список; це стосується і методу find, і функції “\$”; тому, щоб перевірити чи елемент знайдений, потрібно виміряти довжину отриманого списку; в даному рядку ми перевіряємо чи знайшли форму в HTML коді, що прийшов до нас із сервера; у випадку, якщо пост форми був успішний, то, якщо пам'ятаєте, сервер надішле браузеру заголовок про редірект на головну сторінку; тому в цьому випадку HTML код, який отримаємо після AJAX запиту не міститиме форми, а міститиме список студентів; саме тому і потрібна ця умова; під час AJAX запиту обробка редіректів працює таким самим чином як і при звичайному браузерному запиті; таким чином, у функції ‘success’ отримаємо не сторінку з формою, а домашню сторінку нашої аплікації;
- 27ий: якщо форму знайдено, додаємо її до тіла модального вікна; зверніть увагу, що тут, замість методу html, ми скористалися методом append; він додасть код форми до існуючого коду в модальному вікні; інакше, метод html повністю перетре вміст елементу ‘div.modal-body’ тим самим затерши статусне повідомлення, яке ми додали раніше;
- 30ий: новододана форма перетерла стару форму, а разом з нею і усі Javascript обробники, які на неї ми раніше “повішали”; нова форма є абсолютно іншим об’єктом на сторінці і відповідно ще поки жодних Javascript обробників не має; цим рядком ми повторно викликаємо нашу функцію (так, це виклик функції всередині себе самої), щоб ініціалізувати AJAX функціонал для нової форми; передаємо їй першим аргументом об’єкт нової форми, а другим - те саме модальне вікно;
- 36ий: у випадку, якщо форми не знайдено, що означає що форма успішно оброблена, ми перевантажуємо поточну сторінку; вже знайома вам функція reload об’єкта BOM (об’єктної моделі браузера) виконує усю важку роботи з оновлення сторінки; проте ми не виконуємо переван-

таження сторінки одразу, а із затримкою у пів секунди; це для того, щоб користувач міг побачити і прочитати повідомлення про успішне збереження змін в моделі студента; робимо це з допомогою вбудованої функції `setTimeout`³⁰⁸; дана функція приймає першим аргументом анонімну функцію (без назви), яку запустить лише після того як пройде задана кількість мілісекунд; у нашому випадку це 500 мілісекунд, що рівне половині секунди.

Зауважте, що методу `ajaxForm` ми не передали ні адреси куди постити форму, ні інших параметрів, які зазвичай передаємо при виклику метода аjax. Справа у тому, що в даному випадку ми працюємо із формою, а тому плагін `jQuery Form` бере усі дані з форми: URL для запиту із атрибути `action` форми, метод запиту із методу форми, і т.д.

Таким чином, після того, як користувач побачить статусне повідомлення про успішне збереження студента, сторінка оновиться і користувач бачитиме список з нашим оновленим студентом.

Спробуйте перевантажити вашу сторінку і поекспериментувати із оновленою формою. Цього разу запостіть її і гляньте як працює успішне збереження.



За пів-секунди після подібного зображення, сторінка повинна перезавантажуватись

Замість повного перевантаження сторінки можна було б реалізувати оновлення лише потрібного рядка в таблиці, щоб усі зміни на сторінці відбувались ще динамічніше. Оновлений список студентів у вас уже є під руками у вигляді змінної `data` у функції-обробнику 'success' AJAX запиту. Проте даний функціонал залишається вам на самостійне опрацювання. В

³⁰⁸<http://javascript.ru/setTimeout>

такому випадку можна також одразу закривати модальне вікно, а статусне повідомлення показувати на головній сторінці.

...

На завершення нашої з вами роботи над AJAX формою редагування студента поправимо ще один дрібний нюанс.

На даний момент є кілька варіантів того, як можна закрити наше модальне вікно:

- клацнувши по іконці із хрестиком у верхньому правому куті модального вікна;
- натиснувши клавішу Escape;
- клацнувши мишкою де-небудь поза модальним вікном;
- натиснувши кнопку Відмінити;
- натиснувши кнопку Зберегти із правильними даними на формі.

Варіант із клацанням мишкою поза вікном та кнопка Escape можуть часто траплятись випадково. Вони є добре для вікон, коли вікно містить сутінки для ознайомлення, але не дуже добре у випадку роботи із формами. Користувач може потратити кілька хвилин свого часу, щоб правильно заповнити усі дані довгої форми, а тоді випадково клікнути поза вікном, або натиснути на кнопку Escape, не підозрюючи навіть про те, що втратить дані на формі.

Тому, щоб уникнути таких моментів, просто відключимо можливість закривати вікно з допомогою клавіші Escape та кліком мишкою поза вікном. Модальні вікна Twitter Bootstrap уже дають можливість налаштувати вікно відповідним чином. Для цього оновимо виклик методу modal в момент, коли ми показуємо модальне вікно користувачу. У файлі main.js всередині функції initEditStudentPage знаходимо рядок виклику функції modal: "modal.modal('show');", і передаємо даній функції ряд параметрів в словнику:

Забираємо можливість закривати модальне вікно клавішею Escape та кліком мишкої поза вікном

```
1 // setup and show modal window finally
2 modal.modal({
3     'keyboard': false,
4     'backdrop': false,
5     'show': true
6 });

```

В рядку 5 ми викликаємо метод modal і передаємо йому словник із рядом налаштувань:

- ‘keyword’: забороняємо закривати вікно клавішею Escape;
- ‘backdrop’: відключаємо фон позаду модального вікна; без фону місце поза модальним вікном не буде сприймати кліки мишкої;
- ‘show’: показуємо вікно.

Спробуйте востаннє перевантажити сторінку і протестувати можливість із закриванням модального вікна.

Зауважте: якщо браузер не підтримує Javascript (хоча це рідко буває у наші дні), тоді користувач зможе користуватись стандартною формою на окремій сторінці. Без будь-яких проблем. Адже ми не змінили жодний функціонал як на сервері так і на клієнті. Лінк на форму редагування працюватиме як і працював раніше, без приєднаного до нього Javascript обробника. Більшість браузерів підтримують відключення Javascript інтерпретатора, тому можете навіть спробувати і переконатись на власні очі, що форма редагування залишиться працюючою у звичайному режимі.

...

На цьому завершуємо роботу із формою редагування студента. Її можна ще добряче покращити і зробити ще більш інтерактивною. Як? Ознайомтесь із домашніми завданнями, що йдуть далі.

Домашнє завдання 1: На продакшин сервері усі запити будуть трохи довшими і період між кліком по лінку редагування та появою модального вікна із формою може бути доволі довгим. Це змусить користувача засумніватись, що він попав по лінку, або подумати, що лінк взагалі не працючий. Тоді він спробує клікнути ще кілька разів, що може поламатиapplікацію, або перетерти форму в модальному вікні ще раз. Для цього потрібно зробити дві речі: 1) під час запиту на сервер за формую тимчасово відобразити елемент прогресу (spinner, loader) на інтерфейсі користувача, щоб він бачив, що йде робота над його кліком; 2) тимчасово блокувати клік лінків, до завершення AJAX запиту.



Приклад елементу, який показує прогрес при завантаженні пошти на gmail.com. У нашому випадку це може бути статичний текст: "Йде робота..."

Домашнє завдання 2: Подібним чином оновити клік по кнопці Зберегти на формі редагування в модальному вікні, щоб: 1) поля та кнопки форми тимчасово блокувались (скористайтесь атрибуутами `readonly` та `disabled`) для користувача; 2) та щоб показувалось повідомлення поверх форми про те, що триває процес відправки даних на сервер. Підказка: для цього додаєте ваш Javascript код у три параметри AJAX запиту та ajaxForm виклику: '`'beforeSend'`', '`'success'`', '`'error'`'.

Домашнє завдання 3: Пройтись по решті форм у нашійapplікації і оновити їх так, щоб працювали у режимі AJAX.

Думаю багато речей можуть одразу не вдаватись. Спочатку AJAX запити здаватимуться як чорні ящики, де невідомо що відбувається. Тому одразу рекомендую активно використовувати Console закладку плагіна Firebug убраузері Firefox. У ній ви зможете аналізувати повний цикл роботи подібних запитів, ставити точки зупинки Javascript коду і досліджувати необхідні його частини. Це допоможе зрозуміти основну суть роботи із AJAX інтерфейсами.

Домашнє завдання

В даній главі ми розібралися лише з основами Javascript програмування та підходом AJAX для реалізації користувальського інтерфейсу.

Матеріалу даної глави вам вистачить, щоб зрозуміти основний принцип побудови динамічних сторінок. А робота над домашніми завданнями та рішення наблизених до реальних задач заставить вас глибше влізти у нутрощі AJAX запиту та розібратись з тим, що таке зручний для користувача інтерфейс. І, що є найважливішим, ви отримаєте досвід побудови так званих ‘user-friendly’ інтерфейсів та спробуєте поставити себе на місце користувача, адже лише так можна реалізувати по-справжньому зручні і прості у користуванні веб- сайти та applікації.

...

Окрім завдань, які ви знаходили протягом глави, також пропоную спробувати попрацювати над наступним списком завдань середньої складності:

- зробіть, щоб закладки нашої applікації Студенти, Відвідування і т.д. працювали без перезавантаження сторінки;

- так само, щоб через AJAX запити працювало сортування списку студентів;
- також, щоб посторінкова навігація не вимагала перевантаження сторінок.

Зауважте, що при реалізації усіх даних речей вам прийдеться по кілька разів повертатись до попереднього AJAX функціоналу і робити додаткові “фікси”, щоб самостійно прив’язувати Javascript обробники, які для вас прив’язував браузер одразу після звичайного завантаження сторінки.

Тепер значно складніше завдання: будувати можливість використання кнопки браузера Назад і навігувати між станами форми редагування студента та модального вікна загалом.

А якщо детальніше: коли відкриваємо вікно із формою редагування студента, то має можливість знову його закрити кнопкою Назад у браузері.

Для вирішення даної задачі вам потрібно буде самостійно розібратись із ще одним DOM об’єктом: [history³⁰⁹](#) (історія запитів у браузері). У випадку, якщо ми навігуємо сторінками інтернету з допомогою звичайних браузерних запитів, браузер запам’ятовує для нас історію сторінок, які ми переглянули. Таким чином, маємо можливість використовувати кнопку Назад у браузері, щоб повернутись до попередніх сторінок. Проте AJAX запити не записуються в історію, адже перевантажень сторінок не відбувається. Для того, щоб була можливість вертатись до попередніх сторінок та станів при AJAX запитах, з допомогою власного Javascript коду та [цікавих підходів³¹⁰](#) програмісти маніпулюють рядком адреси у браузері.

Це непросте завдання, тому не розчаруйтесь занадто, якщо не виходитиме довгий час.

Ще одне цікаве завдання: спробувати запустити аплікацію в браузері Internet Explorer і перевірити чи наш Javascript код працює. Я сам не тестував, тому якщо знайдете помилки - звітуйте!

І на завершення, бонус-завдання для тих, кому дуже неподобалось те, як виглядало поле “Фото” на наших формах: кастомізувати стандартний Django віджет

³⁰⁹ <http://javascript.ru/window.history>

³¹⁰ <http://habrahabr.ru/post/31211/>

представлення поля зображення. Новий віджет повинен додатково показувати попередній перегляд зображення у невеличкому розмірі. Зробіть візуальну частину на ваш смак.

Завдань досить багато і більшість з них є доволі об'ємними. Той, хто виконає більшість з них, гарантовано зможе зреалізувати більшість динамічних інтерфейсів, що знадобляться йому під час роботи над реальними проектами. А це, думаю, повинно слугувати хорошою мотивацією, щоб присвятити добрячий шмат свого часу для роботи над домашками даної глави.

...

В наступній главі ми з вами тимчасово відійдемо від візуальної частини веб-проекту і зануримось повністю у мову Python та бекенд.

Ознайомимось із сигналами (подіями) в Django та способами ведення лог-файлів в мові Python.

10. Логування дій над студентами: сигнали в Django та Python логер

В цій главі ми з вами розберемось із двома важливими інструментами: події на стороні сервера та ведення журналу (логу) активності аплікації (помилки, статистичні дані, дані з використання аплікації).

Як події так і ведення логів є особливо важливими у великих проектах.

Протягом даної глави ми з вами реалізуємо:

- журнал дій над студентами;
- лог помилок при роботі з формою контакту адміністратора.

З англ. слово *log* перекладається як журнал. Тому я вживатиму даних два слова як взаємозамінні синоніми.

Обидва поняття є досить незвичними для початківця і вимагають добрячого теоретичного вводу перед переходом до практики. Тому перед тим як пereйдемо до реалізації обидвох практичних завдань описаних вище, давайте детально розберемось із наступними речима:

- що таке взагалі події, для чого вони потрібні і коли варто використовувати їх безпосередньо в Django фреймворку;
- що таке лог-файл, для чого він потрібен і як з ним працювати;
- що і коли варто логувати в Python та Django;
- інструменти логування в Python.

Теорія подій

В попередній главі, при роботі з мовою Javascript, ми уже познайомилися із поняттям “події”. Ми мали справу із подіями ініційованими користувачем. В більшості випадків це були кліки мишкою. На кожен з таких кліків по лінку чи кнопці на формі, ми запускали власний Javascript код.

Якщо на стороні браузера події є просто необхідними, то на серверній стороні вони використовуються значно рідше. Взагалі, концепція подій на стороні сервера має зміст лише всередині більших проектів та фреймворків, коли код не є монолітним, а розділеним на окремі компоненти, плагіни, додатки та аплікації.

В таких випадках події використовуються, щоб надати можливість стороннім бібліотекам та аплікаціям впливати на логіку головного коду без прямих змін у ньому. Події є особливо корисними, якщо багато різних кусків коду (аплікацій чи додатків) є зацікавленими в одних і тих самих подіях.

Наприклад, якщо наш фреймворк для нас генерує сторінку, але нам потрібно трохи змінити кінцевий HTML код, тоді подія і її обробник дозволять нам не надписувати всю логіку фреймворка по генерації сторінки, а лише вклинитись в останній момент, щоб внести необхідні корективи. І все це, в такому випадку, можна буде зробити ззовні, без втручання у код фреймворку.

Так само як і у браузері, робота з подіями складається з двох частин:

- в одному місці відбувається подія (ще кажуть: подія ініціюється або “запалюється”);
- в іншому місці (їх може бути багато) є код, який моніторить дану подію і реагує на неї.

Єдина різниця полягає у тому, що на клієнтській стороні браузер ініціює більшість подій (тобто вони є вбудовані), а на сервері події, в основному, “запалюються” фреймворком або нашим власним кодом.

Давайте перейдемо до невеликих прикладів конкретно в Django. Це дозволить нам краще зрозуміти теорію подій і для чого вони взагалі нам потрібні.

Події в Django

Події в Django називаються сигналами (англ. signals). Веб-фреймворк Django містить так званий диспетчер сигналів, який дозволяє різним аплікаціям реагувати на дії, що відбуваються де-інде у фреймворку.

Частина коду, яка запалює подію, називається відправником (англ. sender). А частина коду, яка отримує сигнал, називається отримувачем (англ. receiver). Таким чином, диспетчер подій в Django дозволяє певному набору відправників повідомляти множину отримувачів про те, що певна дія відбулась в системі.

Django надає цілу множину [вбудованих сигналів³¹¹](#), які дозволяють розробникам моніторити багато різноманітних дій у фреймворку. Дані сигнали поділяються на групи:

- [сигнали моделей³¹²](#): pre_save - відсилається перед викликом методу save моделі, post_save - відповідно, після виклику методу save; pre_delete, post_delete - перед і після видалення моделі з бази, і т.д.;
- [сигнали manage.py команд³¹³](#): pre_migrate, post_migrate - відсилаються до і після запуску команди міграції; pre_syncdb і post_syncdb - відсилаються до і після синхронізації моделей з базою даних і т.д.;
- [сигнали роботи із об'єктами запиту та відповіді³¹⁴](#): request_started і request_finished: запускаються на початку та закінченні запиту; got_request_exception - відсилається при помилці на сервері;
- [сигнали при запуску тестів³¹⁵](#): дана група сигналів буде більш зрозумілою нам у наступних главах, коли перейдемо до написання автоматичних тестів на код аплікації;
- [сигнали при роботі з базою даних³¹⁶](#): connection_created - запускається, коли отримано зв'язок з базою.

³¹¹<http://djbook.ru/rel1.7/ref/signals.html>

³¹²<http://djbook.ru/rel1.7/ref/signals.html#module-django.db.models.signals>

³¹³<http://djbook.ru/rel1.7/ref/signals.html#management-signals>

³¹⁴<http://djbook.ru/rel1.7/ref/signals.html#module-django.core.signals>

³¹⁵<http://djbook.ru/rel1.7/ref/signals.html#module-django.test.signals>

³¹⁶<http://djbook.ru/rel1.7/ref/signals.html#module-django.db.backends>

Кожен із сигналів надає, як мінімум, один обов'язковий аргумент для отримувачів сигналу: `sender`. Це об'єкт або клас, від імені якого здійснюється відправка сигналу. В залежності від групи сигналу об'єкт `sender` буде різним. Наприклад, для сигналів моделей `sender` являється класом моделі. Для менеджменту сигналів, `sender` буде інстансом `AppConfig317` класу. Для сигналів пов'язаних із обробкою запитів та підготовкою відповідей (`request/response`), `sender` міститиме посилання на клас обробника запиту (напр. `django.core.handlers.wsgi.WsgiHandler`).

Крім того, кожен тип сигналу має набір своїх власних додаткових параметрів, які він передає функції-отримувачу. Наприклад, сигнал моделі `post_save318` серед додаткових параметрів має:

- `instance`: об'єкт моделі, що був збереженим;
- `created`: булевівська змінна, яка рівна `True` у випадку, коли збережений об'єкт в базі був новостворений;
- `update_fields`: список полів, які потрібно було оновити в об'єкті моделі.

Таким чином, наш обробник події (сигналу) `post_save` матиме усю необхідну інформацію для своєї логіки.

Як реєструвати власний обробник сигналу?

Почнемо з того, що увесь код по роботі із сигналами в Django заведено класти у модуль `signals.py` в корені аплікації.

До Django 1.7 реєстрацію обробників сигналів виконували в модулі або пакеті з моделями. Проте тепер для цього правильним місцем є метод `“ready()”` конфігураційного класу аплікації. Детальніше даний клас і його підключення до реестру ми розглянемо, коли перейдемо до практики.

Сам отримувач (або ще як його можемо називати “обробник”) сигналу є простою Python функцією, що приймає аргумент `sender` та словник необов'язкових параметрів `kwargs`:

³¹⁷<http://djbook.ru/rel1.7/ref/applications.html#django.apps.AppConfig>

³¹⁸<http://djbook.ru/rel1.7/ref/signals.html#post-save>

Приклад простого обробника, що виводить в консоль повідомлення Hello World

```
1 def hello_world_callback(sender, **kwargs):
2     print("Hello World")
```

Зауважте, що варто завжди включати kwargs серед аргументів функції-обробника, адже та сама подія може відсылати різну кількість аргументів взалежності від різноманітних умов. Тому, щоб в деяких випадках обробник не ламався, краще завжди включати kwargs серед аргументів.

Маємо два способи, щоб приєднувати обробник до сигналу:

- з допомогою методу connect об'єкта сигналу;
- з допомогою декоратора receiver.

Ми скористаємось другим способом, щоб приєднати усі необхідні обробники далі під час практичної роботи.

Ось як виглядатиме повний приклад декларації обробника та приєднання його до сигналу request_finished:

Підключаємо наш обробник до сигналу request_finished

```
1 from django.core.signals import request_finished
2 from django.dispatch import receiver
3
4 @receiver(request_finished)
5 def hello_world_callback(sender, **kwargs):
6     print("Hello World")
```

Ми імпортували функцію-декоратор із модуля django.dispatch, а тоді викликали її одразу перед функцією-обробником hello_world_callback. Декоратору передали об'єкт сигналу request_finished в якості аргумента. Таким чином, прив'язавши нашу функцію до даного сигналу.

Декоратор в мові Python - це така собі обгортка функції або метода класу, яка надає додатковий функціонал. Декоратори зручні, коли маємо в багатьох функціях спільний код. З допомогою даного підходу можемо виділити цей спільний код в одну функцію і навішувати його у вигляді декоратора на потрібні функції.

Тут³¹⁹ можете детальніше ознайомитись із концепцією декоратора в Python.

На цьому усе. Більш детально розглянемо на практиці. Але перед тим, як переходити до теорії з логування, варто розглянути випадки коли потрібно, а коли краще утримуватись від використання системи сигналів.

Коли використовувати?

Сигнали в Django є синхронними. Це означає, що обробники (отримувачі) сигналу запускаються в межах того ж процесу, що і основний код, який запускає (відправляє) сигнал. Таким чином, кожен додатковий обробник сигналу буде сповільнювати обробку запиту та формування відповіді клієнту. Напряму додати логіку в код, замість використання сигналу, завжди є кращим з точки зору швидкодії аплікації.

Другою причиною чому не варто зловживати сигналами є те, що з ними код проекту зазвичай стає важчим і заплутанішим для розуміння. Адже там, де в коді відсилається одна подія може відбуватись маса додаткових дій лише завдяки обробникам сигналів. А їх визначати і зрозуміти не так просто, як прямий код.

Саме тому важливо розуміти коли саме використовувати сигналы в Django.

Більшість розробників притримуються спільної думки, що сигналы повинні бути останнім спасінням. Якщо можна обйтись без них, тоді краще обходитись без них.

Ось кілька підказок, коли потрібно використовувати, а коли, навпаки, уникати сигналів:

³¹⁹<http://habrahabr.ru/post/141411/>

- якщо потрібно додати або змінити логіку в Django фреймворку, тоді спочатку перегляньте чи немає іншого способу, ніж прив'язуватись через сигнал; наприклад кастомізнути клас форми чи в'юшки, перекрити шаблон чи URL патерн; Django дуже гнучка система і дозволяє більшість речей перекривати ззовні без необхідності зміни її власного коду і використання сигналів;
- якщо все ж таки немає можливості змінити функціонал Django через кастомізацію, але маєте потрібний вбудований сигнал системи, тоді вже пишете обробник сигналу;
- якщо логіка у вашій власній аплікації повинна надати можливість з інших місць впливати на неї (наприклад з коду в інших аплікаціях), тоді також потрібно розглянути систему сигналів; а саме, створити власний сигнал і розсылати його при потребі;
- в більшості випадків у власному коді можна обходитись без сигналів; лише, якщо потрібно змінювати зовнішній код і він надсилає потрібні для вас сигналі;
- якщо хочете скористатись сигналами моделей, то краще це робити лише, якщо ваш отримувач працює універсально із усіма моделями; якщо ж лише з однією моделлю, то швидше за все можна уникнути використання сигналу і просто кастомізнути потрібний аспект Django фреймворку (валідатор, форму, адмін частину і т.д.).

Як початківцю, у вас, мабуть, не буде гострої потреби часто використовувати сигнали. Особливо тому, що Django дозволяє кастомізувати мало не кожен аспект своєї роботи. Проте тема сигналів є однією з обов'язкових при проходженні тестових завдань для інтерв'ю. Тому рекомендую добре розібратись із даною темою.

...

На завершення цієї невеликої теоретичної секції хочу зауважити, що Django також дозволяє **створювати власні сигналі**³²⁰, а не лише моніторити існуючі.

³²⁰<http://djbook.ru/rel1.7/topics/signals.html#define-signals>

Проте дана тема залишається вам на домашнє опрацювання. Наприкінці глави буде домашнє завдання по створенню власного сигналу.

Теорія логування

Логування - це збереження активності програми у певному узгодженному форматі. Логування придумали та використовують для трьох основних завдань:

- дослідження різного роду помилок та проблем роботи програми на продакшн системах;
- збір та візуалізація статистики роботи користувачів з програмою;
- під час розробки програми для відлову помилок та дебагу коду.

У першому випадку логи дозволяють знаходити причину поточних або потенційних майбутніх помилок в аплікації, можливі діри в безпеці, різного роду атаки та факти неправильного використання програми.

У другому випадку логи дозволяють моніторити як саме програма використовується, будувати тренди та прогнозувати майбутні потреби як по удосконаленню програми так і для кращого задоволення потреб користувачів.

У найпростішому випадку лог - це файл із записами прив'язаними до дати та часу.

Так виглядає лог доступу до аплікації запущеної на сервері Apache

```
1 64.242.88.10 - - [07/Mar/2004:16:05:49 -0800] "GET /twiki/bin/edit/M\
2 ain/Double_bounce_sender?topicparent>Main.ConfigurationVariables HTTP\
3 /1.1" 401 12846
4 64.242.88.10 - - [07/Mar/2004:16:06:51 -0800] "GET /twiki/bin/rdiff/\
5 TWiki/NewUserTemplate?rev1=1.3&rev2=1.2 HTTP/1.1" 200 4523
6 64.242.88.10 - - [07/Mar/2004:16:10:02 -0800] "GET /mailman/listinfo\
7 /hsdivision HTTP/1.1" 200 6291
8 64.242.88.10 - - [07/Mar/2004:16:11:58 -0800] "GET /twiki/bin/view/T\
9 Wiki/WikiSyntax HTTP/1.1" 200 7352
10 64.242.88.10 - - [07/Mar/2004:16:20:55 -0800] "GET /twiki/bin/view/M\
```

```
11 ain/DCCAndPostFix HTTP/1.1" 200 5253
12 64.242.88.10 - - [07/Mar/2004:16:23:12 -0800] "GET /twiki/bin/oops/T\
13 Wiki/AppendixFileSystem?template=oopsmore&m1=1.12&m2=1.12 HTTP/1.1" \
14 200 11382
15 64.242.88.10 - - [07/Mar/2004:16:24:16 -0800] "GET /twiki/bin/view/M\
16 ain/PeterThoeny HTTP/1.1" 200 4924
17 64.242.88.10 - - [07/Mar/2004:16:29:16 -0800] "GET /twiki/bin/edit/M\
18 ain/Header_checks?topicparent>Main.ConfigurationVariables HTTP/1.1" \
19 401 12851
20 64.242.88.10 - - [07/Mar/2004:16:30:29 -0800] "GET /twiki/bin/attach\
21 /Main/OfficeLocations HTTP/1.1" 401 12851
22 64.242.88.10 - - [07/Mar/2004:16:31:48 -0800] "GET /twiki/bin/view/T\
23 Wiki/WebTopicEditTemplate HTTP/1.1" 200 3732
24 64.242.88.10 - - [07/Mar/2004:16:32:50 -0800] "GET /twiki/bin/view/M\
25 ain/WebChanges HTTP/1.1" 200 40520
26 64.242.88.10 - - [07/Mar/2004:16:33:53 -0800] "GET /twiki/bin/edit/M\
27 ain/Smtpd_etrn_restrictions?topicparent>Main.ConfigurationVariables \
28 HTTP/1.1" 401 12851
29 64.242.88.10 - - [07/Mar/2004:16:35:19 -0800] "GET /mailman/listinfo\
30 /business HTTP/1.1" 200 6379
```

У наведеному вище куску лог-файла на кожному рядку маємо IP, час та дату події (тобто доступу до аплікації), тип запиту, адресу запиту, версію HTTP протоколу, статус відповіді та інші дані.

Логи можна оформляти у найрізноманітніших форматах:

- в консолі програми;
- у файли;
- відправляти електронними листами адміністраторам програми;
- HTTP або TCP/IP запитами на віддалений сервер чи веб-сайт;
- і різноманітними іншими каналами передачі даних.

Ось невеликий список груп, з якими приходиться стикатись веб-розробнику. Список поділений згідно призначення лог-файлів:

- *логи веб-сервера*: лог доступу до сервера, лог помилок, лог стану сервера (старт, зупинка, рестарт);
- *логи бази даних*: стан бази, лог запитів у базу даних;
- *логи безпеки сервера*: доступ до сервера (наприклад по протоколу ssh чи ftp), спроби несанкціонованого доступу;
- *системні логи*: функціонування сервера (старт, зупинка, перезавантаження), завантаження сервера (операційної пам'яті, процесора), додаткові логи плагінів для моніторингу комп'ютера;
- *логи роботи мережі*: використання мережі, доступ по мережі, старт і зупинка мережевих сервісів.

Формат кожної з груп логів відрізняється між собою. Але в кожній із груп формати намагаються зберігати однаковими. Це потрібно для того, щоб маючи один і той самий інструмент для аналізу та візуалізації даних лог-файлів можна було, наприклад, працювати із різними веб-серверами. Адже величезні кількості інформації не матимуть особливої цінності без спеціалізованих інструментів для їх аналізу.

Логи можуть містити записи різного пріоритету. Різні групи логів по різному можуть це позначати. Одні записи можуть бути чисто інформативними, інші повідомлення про помилку, а ще інші - критичними поломками програми. Це дозволяє, при потребі, ефективніше пробігатись по записах у пошуках найважливішої інформації в даний момент.

Для моніторингу лог-файлу “вживу” з файлової системи, на Лінуксі є команда “tail -f path/to/file.log”. По мірі заповнення лог-файла новими записами ви їх одразу бачитимете в консолі завдяки команді tail.

З часом логи, які організовуються у файлах, можуть розростатись до величезних розмірів. Вони можуть містити гігабайти інформації. З текстовими файлами, що мають понад сотню мегабайт уже стає складно працювати, а з гігабайтовими просто неможливо, адже для цього необхідна величезна кількість оперативної пам'яті.

Щоб уникати подібних проблем, придумали так званий підхід log rotation. Бу-демо перекладати його як “обертання логів” або ротація логів. Як тільки файл досягає певного максимально встановленого розміру, або як тільки проходить певна фіксована кількість часу, файл переименовується (наприклад з event.log на event.log.1), архівується, а для поточного логу створюється новий порожній файл. Коли головний лог-файл знову час оновлювати на порожній, тоді усі по-передні лог-файли посугаються на одиничку так, що файл event.log.1 перейде у файл event.log.2 і так далі. Таким чином, кожен із таких архівних файлів буде обмеженим у розмірі.

...

Сама концепція логів є доволі простою. Важливі для нас події ми записуємо у файл, а потім, при потребі, переглядаємо дані файли або вручну, або з допомо-гою додаткових інструментів. Проте, як бачите, з вищепереліченої інформації, існує багато нюансів при роботі з логами:

- типи логів;
- формат логів;
- відсилання логів (у файл, консоль, емейл, HTTP протокол);
- пріоритет повідомлень в лог-файлах;
- супорт лог-файлів.

Саме тому тепер, коли перейдемо до інструментів логування в мові Python, прийдеться ознайомитись із чималою кількістю інформації. Адже в Python передбачені інструменти на усі випадки життя. Вони покривають усі вище-перечислені аспекти при роботі з логами, навіть включаючи ротацію великих лог-файлів.

Логування в Python

Початківець, зазвичай, перший час обходиться без Python інструментів для логування. Йому ще не довіряють закидувати проекти на продакшин сервера і відлагоджувати роботу програми для кінцевих користувачів. А під час

роздробки програми і дебагу коду вистачає модуля `pdb`³²¹ (для зупинки коду і дослідження в “живому” режимі) та функції `print`, щоб виводити певні дані програми у робочу консоль.

В простіших випадках функції `print` цілком вистачає при розробці. Але у деяких випадках це може виливатись у забуті `print` виклики в продакшин коді. Які, в свою чергу, сповільнюють роботу програми і можуть навіть її повністю припинити. Крім того, `print` виводить інформацію лише у консоль, яка досить швидко зникає і не архівується.

Саме тому в мові Python є спеціальний окремий інструмент для логування. Називається він модулем `logging`³²².

Даний модуль надає ряд функцій для логування: `debug`, `info`, `warning`, `error` і `critical`. В залежності від цілі лог-запису та його пріоритету ми вибираємо необхідну функцію з модуля `logging`.

Ось кілька детальніших підказок, коли використовувати кожну із функцій:

- функція `print`: якщо тимчасово треба швидко потестувати змінні середовища в процесі розробки; головне не забувати потім забирати дані виклики після дебагу коду;
- `info` та `debug`: для виводу подій щодо звичайного функціонування програми, як моніторинг статусу програми чи доступу до неї; для діагностичних цілей;
- `warning`: для виводу повідомень про покращення коду чи налаштувань програми; також для попереджень про майбутні проблеми при міграції проекту на нову версію фреймворку;
- `error`, `exception`: для звіту про помилку; при цьому помилка може показуватись користувачу або ні; в будь-якому разі дані функції запишуть повний текст помилки в лог і, таким чином, дозволять розробнику розібратись із проблемою;
- `critical`: для виводу критичних для програми збоїв; зазвичай помилки несумісні із запуском програми, або такі, що відключають окремий великий функціонал програми.

³²¹<http://habrahabr.ru/post/104086/>

³²²<https://docs.djangoproject.com/en/1.7/topics/logging/>

Дані функції логування називаються відповідно до серйозності подій, які вони логують. В мові Python є наступні рівні подій (в дужках наведено цифрове значення кожного із рівнів):

- DEBUG (10): інформація корисна, здебільшого, при діагностуванні проблем;
- INFO (20): підтвердження того, що все йде як потрібно;
- WARNING (30): все працює добре, але є незначна проблема, або вона може виникнути незабаром;
- ERROR (40): через доволі серйозну проблему програма не змогла виконати певну функцію;
- CRITICAL (50): проблема, через яку програма не може працювати.

В Python дефолтним рівнем логування є рівень WARNING. Це означає, що повідомлення цього рівня і вище (тобто важливіші) будуть логуватись.

Тепер давайте пройдемось по кількох простих прикладах, щоб зрозуміти принцип роботи із logging модулем. А після цього вже розглянемо решту можливостей цього надзвичайно потужного модуля.

Простий приклад

Давайте скористаємося кількома функціями модуля logging і на простому прикладі продемонструємо принцип логування:

Логуємо повідомлення в консоль

```
1 import logging
2
3 logging.warning('Hello World!')
4 logging.info('Test Message')
```

Збережіть вищезгаданий кусок коду де-небудь у файловій системі у новому файлі. Запустіть його і ось, що отримаєте:

Результат скрипта логування

```
1 WARNING:root>Hello World!
```

Ми отримали лише перше повідомлення. Дефолтний рівень логування в Python є INFO, тому усі повідомлення з нижчим рівнем важливості ігноруються. Цей дефолтний рівень логування можна змінювати в контексті кожного окремого логера. Але це ми розглянемо трохи пізніше.

Як бачите, наші функції `warning` і `info` дуже прості у використанні. Приймають стрічку для виводу і все. Проте, вони також можуть приймати додаткові змінні, які потрібно вклести у стрічку. Це подібне до інтерполяції стрічок. Ось приклад:

Логування значень змінних

```
1 import logging
2
3 age = 10
4 name = 'Roman'
5
6 logging.warning('%s is %d years old.', name, age)
```

Збережіть даний код у інший файл і запустіть його інтерпретатором Python. Ось результат:

Результат змінних в лог-записі

```
1 WARNING:root:Roman is 10 years old.
```

Змінні були підставлені у стрічку в потрібних місцях.

Зверніть увагу на формат повідомлення:

- `WARNING`: пріоритет повідомлення;

- root: використовуємо головний дефолтний логер; ми повернемось до цього пізніше;
- “Roman is 10 years old.”: текст повідомлення.

Даний формат ми можемо також змінювати. Як саме? Розберемо далі.

Логування у файл

Найбільш поширеним сховищем для логів є файли у файловій системі. Тому давайте спробуємо швидкий приклад, який заставить лог-записи потрапляти не в консоль, а у файл:

Логуємо у файл

```
1 import logging
2
3 # configure file logging
4 logging.basicConfig(filename='mytest.log', level=logging.DEBUG)
5
6 logging.debug('We log into file')
7 logging.info('INFO message we log there too')
8 logging.warning('WARNING we log into file as well!')
```

У вищеприведеному прикладі маємо використання нової для нас функції basicConfig. Даної функції конфігурує дефолтний логер, який ми поки постійно використовуємо. Ми налаштували два параметри:

- filename: адресу файлу, куди записувати повідомлення;
- level: мінімальний пріоритет повідомлень, які потрібно брати до уваги.

В результаті запуску даного скрипта ви отримаєте новий файл mytest.log. Він знаходитиметься в тій директорії, де ви запускали ваш Python скрипт. Ось, що знайдете в даному файлі:

mytest.log

```
1 DEBUG:root:We log into file
2 INFO:root:INFO message we log there too
3 WARNING:root:WARNING we log into file as well!
```

Форматування лог-записів

Також можемо впливати на формат повідомлень в лозі. Це робимо з використанням тієї самої функції basicConfig, але цього разу передаємо новий аргумент format.

При форматуванні маємо цілий ряд ключових слів, які дадуть нам доступ до різноманітних аспектів повідомлення:

- **levelname**: назва рівня повідомлення;
- **message**: текст повідомлення;
- **asctime**: дата та час повідомлення;
- **module**: назва модуля звідки відбувається логування;
- і ще кілька інших опцій.

Ось яким чином можна впливати на формат повідомлень:

Змінюємо формат лог-повідомлень

```
1 import logging
2
3 logging.basicConfig(format='%(asctime)s %(levelname)s:%(message)s',
4     level=logging.DEBUG)
5
6 logging.debug('This is message with custom format')
7 logging.info('And this one too')
```

Після запуску даного скрипта ви отримаєте наступні повідомлення в командній стрічці:

Форматовані записи

```
1 2015-02-09 12:24:54,444 DEBUG:This is message with custom format
2 2015-02-09 12:24:54,444 INFO:And this one too
```

Як бачите, маємо на початку дату та час повідомлення, далі пріоритет і, на завершення, текст самого повідомлення.

...

Це є необхідний мінімум знань, щоб працювати із логуванням. Проте, у більших проектах варто використовувати не головний дефолтний Python логер, а свої власні. Це дасть можливість гнучкішої конфігурації логування для кожного окремого модуля аплікації, а також мати містилище логів в різних місцях.

Модуль `logging` дає нам такі поняття, як:

- логери (`loggers`): об'єкти, які виконують основну роботу з логування;
- обробники (`handlers`): об'єкти, які передають і зберігають повідомлення у потрібному місці;
- компоненти форматування (`formatters`): дозволяють визначати кастомні формати для повідомень;
- фільтри (`filters`): визначають чи пропускати повідомлення чи ігнорувати; дані компоненти ми не будемо розглядати; їхнє використання не настільки поширене, тому залишається на самостійне опрацювання.

Давайте трохи детальніше оглянемо дані компоненти і, наприкінці, розберемо варіанти для їхньої кінцевої конфігурації.

Логери (Loggers)

Коли ми використовували методи `basicConfig`, `info`, `error` з модуля `logging`, ми, насправді, використовували методи глобального дефолтного логера. В проекті ми створюватимемо окремі логери для різних цілей.

Логер виконує три функції:

- надає методи для логування (info, error, debug і т.д.);
- вирішує, які повідомлення ігнорувати, а які логувати, базуючись на поточно встановленому рівні пріоритетів;
- передає необхідні повідомлення потрібним обробникам.

Більшість методів, якими ми користуватимемось належать до двох категорій:

- конфігураційні: setLevel, addHandler, addFilter;
- відсилання повідомлень: exception, info, debug і т.д.

Щоб створити логер, потрібно скористатись функцією getLogger модуля logging і передати їй ім'я для логера. Рекомендується використовувати імена модулів для імен логерів. Логери підтримують унаслідування таким чином, що логер, який створений під іменем students.views.students_list унаслідуватиме усі налаштування логера під іменем students.views.

Унаслідування означає, що будь-які налаштування, яких бракує на поточному логері братимуться автоматично із батьківського логера. І такий ланцюжок дійде аж до дефолтного логера, якщо потрібного налаштування не було знайдено в попередніх логерах.

Також повідомлення, які перехоплює логер доходять і до батьківських логерів (це можна заборонити опцією propagate). Тому немає потреби створювати обробники для усіх логерів, а лише для кореневого. А обробники для вкладених логерів можна конфігурувати пізніше і лише при потребі.

Далі в секції конфігурації ми оглянемо повний приклад створення і конфігурації логера.

Обробники (Handlers)

Аплікація може мати потребу у відсиланні повідомлень про події у різні системи. Наприклад:

- доступ до аплікації: лог файл + консоль;
- критичні помилки на продакшин сайті: емейлом адміну;
- повідомлення про можливі майбутні проблеми: у зовнішню систему звітності для розробників через HTTP протокол або REST API.

Таким чином, повідомлення може потрапляти одразу у кілька різних місць для зберігання, або взагалі бути проігнорованим.

Обробники, так само як і логери, можуть мати свій встановлений рівень (приоритет) для повідомлень, на які вони реагують. Це дозволяє більш гнучко конфігурувати роботу із повідомленнями.

Також обробники мають власний формат повідомлень і, при потребі, фільтри. Таким чином, об'єкт обробника володіє наступними методами:

- `setLevel`: щоб змінювати рівень повідомлень;
- `setFormatter`: щоб встановлювати об'єкт компоненти форматування;
- `addFilter`, `removeFilter`: робота з об'єктами фільтрами.

Після створення об'єкта обробника важливо напряму встановити його одному або більше логерам. Інакше, ефекту від такого обробника не буде.

Python logging модуль надає велику кількість обробників:

- `StreamHandler`: надсилає повідомлення до файло-подібних потоків (як консоль, вивід для помилок і т.д.);
- `FileHandler`: надсилає повідомлення до файлів на диску;
- `BaseRotatingHandler`, `RotatingFileHandler`, `TimedRotatingFileHandler`: надсилає повідомлення до файлів на диску, а також займається їхньою ротацією;
- `SocketHandler`: до TCP/IP сокетів;
- `SMTPHandler`: надсилає повідомлення на емейл адреси;
- `SysLogHandler`: до системних логів *nix систем;
- `NTEventLogHandler`: до системних логів Windows-подібних машин;
- `MemoryHandler`: зберігає логи в оперативній пам'яті;
- `HTTPHandler`: з допомогою методів GET і POST надсилає повідомлення через HTTP протокол;
- `NullHandler`: порожній обробник, який шле повідомлення в нікуди;
- і ще багато інших обробників.

В секції конфігурації розглянемо код з прикладами створення і конфігурації обробників.

Форматування (Formatters)

Навіть для зміни форматування лог-повідомлень у Python є окремий клас: `Formatter`.

Об'єкти даного класу формують структуру та формат окремих повідомлень в лозі. На практиці вистачає користуватись даним класом, щоб створювати необхідні формати повідомлень.

На етапі ініціалізації даний клас приймає стрічку із доступними параметрами для конфігурації. Дані параметри ми вже оглянули з вами у секції із швидкими прикладами.

Маючи один об'єкт форматування, його можна присвоювати різним обробникам і логерам. Так само, як і у випадку з обробником, без прив'язки об'єкта форматування до логера, користі з нього ніякої.

Давайте тепер перейдемо до конфігурації усіх цих компонент разом.

Конфігурація логування

Програмісти можуть конфігурувати логування наступними трьома шляхами:

- з Python коду використовуючи методи об'єктів описаних вище;
- з допомогою окремого конфігураційного файлу і передаючи його функції `fileConfig`;
- створюючи конфігураційний Python словник і передаючи його функції `dictConfig`.

Для прикладу, щоб краще розуміти структуру і поєднання вищеописаних об'єктів, ми скористаємося першим способом. В Django користуються третім способом, щоб конфігурувати логери для проекту.

Перейдемо одразу до прикладу конфігурації нашого власного логера із компонентою обробника та форматування. Найважливіші рядки наведені з коментарями:

Конфігуруємо логування з Python коду

```
1 # -*- coding: utf-8 -*-
2 import logging
3
4 # створюємо логер, ім'я передаємо __name__
5 # дана вбудована змінна вказує на ім'я поточного модуля
6 logger = logging.getLogger(__name__)
7
8 # встановлюємо пріоритет повідомлень в DEBUG
9 # модуль logging містить константи для усіх рівнів
10 logger.setLevel(logging.DEBUG)
11
12 # створюємо консольний обробник і встановлюємо йому
13 # рівень логування DEBUG повідомлень
14 ch = logging.StreamHandler()
15 ch.setLevel(logging.DEBUG)
16
17 # створюємо компоненту по форматуванню
18 formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)\\
19 s - %(message)s')
20
21 # задаємо формат для нашого обробника консолі
22 ch.setFormatter(formatter)
23
24 # ну і не забуваємо наш обробник встановити як
25 # обробник нашого власного логера
26 logger.addHandler(ch)
27
28 # а тут уже відбувається саме логування
29 logger.debug('debug повідомлення')
30 logger.info('info повідомлення')
31 logger.warn('warn повідомлення')
32 logger.error('error повідомлення')
33 logger.critical('critical повідомлення')
```

Після запуску вищеприведеного коду отримаєте наступний результат:

Результат запуску скрипта з логуванням

```
1 2015-02-09 13:33:13,997 - __main__ - DEBUG - debug повідомлення
2 2015-02-09 13:33:13,997 - __main__ - INFO - info повідомлення
3 2015-02-09 13:33:13,998 - __main__ - WARNING - warn повідомлення
4 2015-02-09 13:33:13,998 - __main__ - ERROR - error повідомлення
5 2015-02-09 13:33:13,998 - __main__ - CRITICAL - critical повідомлення
```

Зауважте, що ми отримали “`__main__`” в якості назви модуля. Справа в тому, що ми напряму запустили скрипт, тому змінна “`__name__`” вказує не на ім’я файлу, а на ключове слово “`__main__`”. У випадку, коли будемо імпортувати код модуля, а не напряму запускати сам модуль, отримуватимемо назву модуля під даною вбудованою змінною “`__name__`”.

Давайте розглянемо ще один приклад. В даному випадку продемонструємо поширені шаблон обробки помилок, коли помилка не відображається на користувачький інтерфейс, а логується для подальшого розбору програмістами:

Логування помилок

```
1 import logging
2
3 try:
4     open( '/path/to/file/does/not/exist' , 'rb' )
5 except Exception, e:
6     logging.error( 'Failed to open file' , exc_info=True)
```

У випадку, коли відбувається помилка (у нашому випадку ми пробуємо відкрити файл, якого не існує), отримуємо помилку. Перехоплюємо її і логуємо. Зауважте, що у даному випадку ми передали додатковий аргумент функції `error: exc_info`. Даним аргументом ми вказали функції додати повний текст помилки до нашого кастомного повідомлення. Кожен із рапортуючих методів логера приймає даний аргумент `exc_info`.

Якщо спробуєте запустити даний код, тоді отримаєте щось подібне до наступного:

Повідомлення про мілку разом із повним кодом помилки

```
1 ERROR:root:Failed to open file
2 Traceback (most recent call last):
3   File "log.py", line 4, in <module>
4     open('/path/to/does/not/exist/file', 'rb')
5 IOError: [Errno 2] No such file or directory: '/path/to/does/not/exi\
6 st/file'
```

Можемо бачити наше повідомлення “ERROR:root:Failed to open file”, а також текст помилки, що слідує за ним. Без нього програмісту буде значно важче відловити і знайти причину помилки.

...

Як бачите, інформації дуже багато, адже модуль `logging` надає дуже гнучкий фреймворк для роботи з логами. Врахований практично кожен аспект логування.

Насправді, на практиці все значно простіше. Як тільки ви відконфігурували свої логери, саме логування відбувається надзвичайно просто з допомогою виклику однієї із функцій модуля `logging`.

На завершення теоретичної частини логування в мові Python хочу зауважити, що усі повідомлення в програмі існують у вигляді об'єктів класу `LogRecord`³²³.

А тепер давайте перейдемо до логування в контексті Django фреймворку. Він підготував для нас приємний сюрприз у вигляді спрощеної конфігурації логерів.

Логування в Django

Django веб-фреймворк повністю перевикористовує Python модуль логування. Тому все те, що ми з вами вивчили у попередній частині даної глави залишається дійсним і в контексті Django.

³²³<https://docs.python.org/2/library/logging.html#logging.LogRecord>

Django використовує варіант конфігурації логування через Python словник. Його ми налаштовуємо в модулі проекту `settings.py` у змінній під назвою `LOGGING`³²⁴.

По-замовчуванню, Django для нас конфігурує кілька логерів, включно з `django.request` логером. Завдяки цьому маємо логування помилок в консоль та на сторінку браузера, якщо працюємо в DEBUG режимі. На продакшині усі повідомлення з пріоритетом ERROR та CRITICAL відсилаються емейлом адміністратору сайту.

Ми відключимо усі дефолтні логери і продублюємо частину у нашій конфігурації.

На завершення великої теоретичної частини таки зробимо одну практичну річ - повністю налаштуємо усі необхідні нам логери.

Ось повний список вимог щодо логування:

- по-перше, дублюємо дефолтний Django логер, який ловить усі повідомлення і ігнорує їх; він спрацьовує тоді, коли повідомлення не було оброблене жодним із інших логерів;
- організовуємо логер для модуля сигналів, який буде логувати повідомлення рівня INFO і вище у два місця: консоль та файл;
- готуємо ще один логер, який буде логувати повідомлення з модуля відправки емейла адміну; будемо перехоплювати помилки при відправці і логувати їх у лог файл;
- відповідно, маємо також підготувати компоненту форматування та обробники для нашого лог-файлу;
- форматувати будемо повідомлення у вигляді довшого формату, де включаємо і дату, і назву модуля, і рівень повідомлення.

Відкриваємо модуль `settings.py` всередині нашого проекту `studentsdb` і додаємо наступні дві змінні `LOG_FILE` та `LOGGING`:

³²⁴<https://docs.djangoproject.com/en/1.7/ref/settings/#std:setting-LOGGING>

Налаштовуємо логери для нашої аплікації

```
1 LOG_FILE = os.path.join(BASE_DIR, 'studentsdb.log')
2
3 LOGGING = {
4     'version': 1,
5     'disable_existing_loggers': True,
6     'formatters': {
7         'verbose': {
8             'format': '%(levelname)s %(asctime)s %(module)s: %(message)s'
9         },
10        },
11        'simple': {
12            'format': '%(levelname)s: %(message)s'
13        },
14    },
15    'handlers': {
16        'null': {
17            'level': 'DEBUG',
18            'class': 'logging.NullHandler',
19        },
20        'console': {
21            'level': 'INFO',
22            'class': 'logging.StreamHandler',
23            'formatter': 'verbose'
24        },
25        'file': {
26            'level': 'INFO',
27            'class': 'logging.FileHandler',
28            'filename': LOG_FILE,
29            'formatter': 'verbose'
30        },
31    },
32    'loggers': {
33        'django': {
34            'handlers': ['null'],
```

```
35         'propagate': True,
36         'level': 'INFO',
37     },
38     'students.signals': {
39         'handlers': ['console', 'file'],
40         'level': 'INFO',
41     },
42     'students.views.contact_admin': {
43         'handlers': ['console', 'file'],
44         'level': 'INFO',
45     }
46 }
47 }
```

Давайте детально розглянемо конфігурацію логування у нашому проекті:

- 1ий рядок: у змінну LOG_FILE ми задаємо шлях до файлу, де міститимуться усі наші лог-повідомлення; лог-файл знаходитиметься в корені папки репозиторію, але ми також додамо даний файл до списку ігнорування; лог-файл є автогенерованим файлом, а значить не повинен міститись всередині репозиторію коду;
- 3ій: декларуємо змінну LOGGING, яка міститиме усю інформацію по логерах в проекті; дана змінна є Python словником;
- 4ий: поки перша і єдина версія формату dictConfig у Python logging модулі: 1;
- 5ий: ключ ‘disable_existing_loggers’ вимикає дефолтні Django логери і налаштування;
- 6ий: ключ ‘formatters’ містить конфігурацію компонент форматування; у нашому випадку визначаємо дві таких компоненти;
- 7ий: одну назовемо ‘verbose’ і використовуватимемо, щоб логувати довший (багатослівний) формат повідомлення;
- 8ий: ключ ‘format’ вказує на формат повідомлення; ми вже з ним озномились і знаємо, що означають дані стрічки інтерполяції в стрічці формату;

- 11ий: інша компонента логуватиме коротший формат повідомлень і називатиметься ‘simple’; включає лише номер рівня та текст повідомлення у тілі лог-запису;
- 15ий: під ключем ‘handlers’ описуємо набір обробників, які пізніше приєднаємо до кількох наших кастомних логерів;
- 16ий: обробник ‘null’ посилаємо повідомлення в нікуди; по-суті, ігноруватиме їх; він використовуватиметься дефолтним Django логером, який ми переб’ємо далі в словнику;
- 17ий: рівень у нього ‘DEBUG’; тобто ловить абсолютно все;
- 18ий: даний обробник йде разом з модулем logging;
- 20ий: наступний обробник - це ‘console’ і він логуватиме повідомлення в командну стрічку, якщо аплікація запущена у так званому режимі переднього плану (англ. foreground); даний обробник ловить все що ‘INFO’ та вище; клас обробника також знаходиться в logging модулі і називається він StreamHandler;
- 23ій: через ключ ‘formatter’ ми прив’язуємо попередньо визначену компоненту форматування до поточного обробника; в нашому випадку ми використовуємо ‘verbose’ формат;
- 25ий: і останній обробник буде логувати повідомлення у файл; називаємо його ‘file’ і передаємо йому клас обробника знову ж таки з модуля logging: FileHandler; формат також ‘verbose’; зауважте додатковий параметр ‘filename’; він вказує на шлях, де знаходиться файл для логування; дану змінну ми з вами визначили перед змінною LOGGING;
- 32ий: починаємо секцію конфігурації найголовніших компонент - логерів; маємо їх аж три;
- 33ій: дефолтний логер ‘django’, який ловитиме усі повідомлення, не зловлені іншими логерами;
- 34ий: ключ ‘handlers’ може містити список обробників; тут можна перелічити усі обробники, які ми визначили напередодні у словнику LOGGING; в даному випадку використовуємо ‘null’;
- 35ий: ключ ‘propagate’ вказує на те, чи повідомляти батьківські логери про повідомлення; по-замовчуванню, даний параметр є True;
- 36ий: так само як і обробник, логер має рівень повідомлень; ‘django’ логер ловить все - рівень ‘INFO’;

- 38ий: визначаємо наш кастомний логер ‘students.signals’; усі повідомлення, що генеруватимемо зсередини модуля signals в корені аплікації надсилаємо використовуючи даний логер;
- 39ий: даний логер надсилаємо повідомлення у консоль та у файл одночасно; у консоль, щоб було зручно під час розробки; у лог, щоб можна було на продакшині досліджувати статистику роботи аплікації;
- 42ий: ще один кастомний логер використовуватимемо для логування помилок при відправці листа з форми контакту адміністратора; називаємо логери згідно шляху до модуля, в яких вони використовуватимуться: ‘students.views.contact_admin’; цей логер також надсилаємо повідомлення у файл та консоль і перехоплюватиме повідомлення усіх пріоритетів.

Якщо збережете ваш модуль settings.py і зробите рестарт Django сервера, нові логери стануть для вас доступними з коду.

Таким чином, маємо усе необхідне, щоб переходити до логування самих повідомлень. А почнемо із реалізації журналу повідомлення щодо подій із студентами.

Ведемо журнал дій над студентами

В даній секції реалізуємо журнал дій над студентами. Це буде файл, який міститиме записи про усі дії над студентами, а саме: додавання, редагування та видалення студентів. Кожен запис про дію міститиме:

- дату та час події;
- рівень події (INFO, DEBUG, і т.д.);
- модуль, з якого логувалась подія;
- тип події (редагування, створення чи видалення);
- повне ім’я та ID студента.

Подібного роду журнали дій корисно мати у випадку, якщо потрібно дослідити чому база даних саме в такому стані, а також, коли потрібно з’ясувати хто і коли робив певні дії з тим чи іншим студентом. Звичайно, журнали подібно роду також корисні при досліженні помилок на продакшині.

Обробник дій над студентами

Почнемо з того, що створимо новий модуль в кореніapplікації:

Створюємо модуль signals.py

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students
2 $ touch signals.py
```

Відкриваємо новостворений модуль signals.py в редакторі коду і додаємо функцію під назвою log_student_updated_added_event, яка оброблятиме додавання та редагування студента:

Заготовка обробника збереження моделі Student в базу

```
1 from django.db.models.signals import post_save
2 from django.dispatch import receiver
3
4 from .models import Student
5
6
7 @receiver(post_save, sender=Student)
8 def log_student_updated_added_event(sender, **kwargs):
9     print sender
```

Більшість коду вам уже зрозуміла базуючись на теоретичній частині даної глави. Лише нагадаю, що у даному випадку працюємо з об'єктом сигналу post_save. Він викликається кожного разу після виклику метода save() на об'єкті моделі. Тобто, після збереження моделі в базу.

Функцію обробника прив'язуємо до сигналу post_save з допомогою декоратора receiver. Крім типу сигналу, даний декоратор отримує другим параметром ключовий аргумент sender. Він вказує на те, що ми зацікавлені лише в save() методі викликаному на моделі студента. Якщо б ми не передали даного аргументу декоратору, тоді б наш обробник викликався при збереженні усіх типів моделей в проекті.

В Django є сигнал `post_save`, який виконується одразу після виклику методу `save()` на об'єкті моделі. Тобто одразу після збереження даних у базу. Даний сигнал працює і після створення нового об'єкту, і після редагування існуючого. Тому ми назвали функцію `_updated_added_` іменем, що означає, що дана функція відповідає за обидві події. Серед аргументів дана функція отримає булеванівську змінну `'created'`, яка служить індикатором новоствореного об'єкта. Ним ми і скористаємося далі.

На даному етапі ми лише викликаємо функцію `print` всередині нашого обробника і хочемо глянути, що ж передається під аргументом `sender`.

Якщо спробуєте поредагувати студента, то не отримаєте жодних повідомлень у вашій консолі (там, де ви запустили команду `runserver`). Справа в тому, що Django ще поки нічого не знає про наші обробники сигналу. Щоб спрацював декоратор `receiver`, потрібно, щоб модуль `signals.py` був імпортований в процесі роботи сервера.

Підключаємо наші обробники сигналів в Django

До версії 1.7 сигнали в Django реєструвались всередині модулів з моделями. Проте з приходом Django 1.7 ситуація змінилась.

Будь-яка Django аплікація реєструється з допомогою конфігураційного класу. Але ми, як розробники, нічого про нього не знаємо до моменту, поки нам не потрібно його кастомізнути. Django створює і конфігурує даний конфігураційний клас за нас. Починачи з Django 1.7 даний клас має додатковий метод під назвою `ready`³²⁵. Цей метод дозволяє запускати код, коли основний функціонал Django (включно із системою сигналів) уже є запущеним та готовим до використання.

Можете думати про даний метод `ready()`, як про одноіменний jQuery метод, який дозволяє запускати власний Javascript код лише, коли браузер згенерував

³²⁵<http://djbook.ru/rel1.7/ref/applications.html#django.apps.AppConfig.ready>

для нас DOM. Подібним чином ми імпортуватимемо модуль `signals` із середи- ни метода `ready`. В цей момент система Django сигналів уже буде налаштова- ною і наші обробники коректно прив'яжуться до потрібних сигналів.

Для цього потрібно, в першу чергу, створити модуль `apps.py` всередині апліка- ції і додати в ньому клас для нашої кастомної конфігурації аплікації студентів:

Створюємо модуль `apps.py` всередині аплікації `students`

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students
2 $ touch apps.py
```

Тепер відкриваємо новостворений модуль в редакторі і прописуємо насту- пний код:

Клас кофігурації аплікації `students`

```
1 # -*- coding: utf-8 -*-
2 from django.apps import AppConfig
3
4
5 class StudentsAppConfig(AppConfig):
6     name = 'students'
7     verbose_name = u'База Студентів'
8
9     def ready(self):
10         from students import signals
```

Давайте порядково розберемось із нашим класом:

- 1ий рядок: метастрічка для декларації кодування `utf-8`; вона є обов'язковою, оскільки далі в коді використовуватимемо кириличний текст; в одній з наступних глав, коли розбиратимемось із перекладами, позбудемось кирилиці у наших Python файлах;

- 2ий: імпортуємо клас `AppConfig326`, який використаємо для унаслідування; він дасть основний функціонал, який ми лише трохи перекриємо у власному класі;
- 5ий: декларуємо клас конфігурації аплікації і унаслідуємось від базового класу `AppConfig`; назва класу може бути довільною, але краще щоб відповідала призначенню; у нашому випадку - це клас конфігурації аплікації `students`; відповідно і назва класу `Students AppConfig`;
- 6ий: обов'язковий атрибут класу `name`; служить для унікальної ідентифікації аплікації в системі;
- 7ий: атрибут `verbose_name` не є обов'язковим і до цього моменту Django генерував його для нас на базі назви аплікації; ви могли бачити стрічку 'students' усюди в адміністративному інтерфейсі Django; тепер там буде повноцінний поясннювальний текст;
- 9ий: метод, заради якого ми, власне, кастомізували дефолтний клас аплікації; даний метод викличеться, коли усі необхідні налаштування системи Django вже відбудуться;
- 10ий: оскільки ми використовуємо декоратори для прив'язки обробника до сигналу все, що нам потрібно зробити - це імпортувати модуль `signals`; при імпорті інтерпретатор запускає код кореневого рівня модуля; відповідно, виклик декораторів вступить в дію і зареєструє наші обробники.

Ми з вами щойно створили новий модуль `apps.py` і наповнили його власним Python класом. Але в Django даний модуль не вичитується автоматично. Про існування даного класу ще потрібно повідомити Django фреймворк. Це робимо з допомогою змінної `default_app_config` в ініціалізаційному модулі пакету в корені аплікації:

`students/init.py`

```
1 default_app_config = 'students.apps.StudentsAppConfig'
```

Ми просто визначили змінну `default_app_config` і присвоїли їй стрічку, яка представляє шлях до нашого кастомного конфігураційного класу аплікації:

³²⁶<http://djbook.ru/rel1.7/ref/applications.html#application-configuration>

‘students.apps.StudentsAppConfig’. На етапі ініціалізації нашої аплікації Django запустить `__init__.py` модуль і вичитає звідти дану змінну.

Лише тепер можете перезавантажити ваш Django сервер і спробувати передавувати студента. Приблизно такий результат повинен бути у вашій консолі:

```
System check identified no issues (0 silenced).
February 15, 2015 - 16:56:25
Django version 1.7.1, using settings 'studentsdb.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
[15/Feb/2015 16:56:27] "GET /students/1/edit/ HTTP/1.1" 200 7427
<class 'students.models.students.Student'>
[15/Feb/2015 16:56:29] "POST /students/1/edit/ HTTP/1.1" 302 0
[15/Feb/2015 16:56:29] "GET /?status_message=%D0%A1%D1%82%D1%83%D0%B4%D0%BD%D0%BD%D0%82%D0%83%D0%82%D0%80%D1%83%D1%81%D0%BF
%D1%96%D1%88%D0%BD%D0%BE%D0%80%D0%8B%D0%85%D0%80%D0%85%D0%86%D0%85%D0%85%D0%BD%D0%BE! HTTP/1.1" 200 7975
[15/Feb/2015 16:56:29] "GET / HTTP/1.1" 200 7854
[15/Feb/2015 16:56:30] "GET /media/podoba3.jpg HTTP/1.1" 304 0
[15/Feb/2015 16:56:30] "GET /static/css/main.css HTTP/1.1" 304 0
[15/Feb/2015 16:56:30] "GET /static/js/jquery.cookie.js HTTP/1.1" 304 0
[15/Feb/2015 16:56:30] "GET /static/js/main.js HTTP/1.1" 304 0
[15/Feb/2015 16:56:30] "GET /media/me_A9A8xdR.jpeg HTTP/1.1" 304 0
[15/Feb/2015 16:56:30] "GET /static/img/default_user.png HTTP/1.1" 304 0
```

Вивід об'єкта sender обробником сигналу

Як бачите, об'єкт `sender` виявився не чим іншим, як класом моделі, яку ми передали аргументом в декоратор `receiver`.

Також зауважте, що тепер в адмінці наша аплікації називається “База Студентів”:

База Студентів		
Групи	Додати	Змінити
Місячні Журнали	Додати	Змінити
Студенти	Додати	Змінити

Результат кастомної конфігурації аплікації

Логуємо події у файл

Тепер, коли ми базово розібралися із сигналами та підключили наш обробник у загальну систему Django сигналів, можемо переходити до логування подій у файл.

Ми уже налаштували логери в модулі `settings.py`, тому в нашему обробнику нам залишається скористатись ними і писати повідомлення у файл.

Ось оновлений код функції `log_student_updated_added_event`:

Логуємо додавання та редагування студента у файл

```
1 # -*- coding: utf-8 -*-
2 import logging
3
4 from django.db.models.signals import post_save
5 from django.dispatch import receiver
6
7 from .models import Student
8
9
10 @receiver(post_save, sender=Student)
11 def log_student_updated_added_event(sender, **kwargs):
12     """Writes information about newly added or updated student into \
13 log file"""
14     logger = logging.getLogger(__name__)
15
16     student = kwargs['instance']
17     if kwargs['created']:
18         logger.info("Student added: %s %s (ID: %d)", student.first_n\
19 ame,
20                 student.last_name, student.id)
21     else:
22         logger.info("Student updated: %s %s (ID: %d)", student.first\
23 _name,
24                 student.last_name, student.id)
```

Давайте детальніше пройдемось по нових для нас рядках:

- 2ий рядок: імпортуємо Python модуль для логування повідомень;

- 14ий: отримуємо логер під іменем поточного модуля (вбудована змінна `__name__` дає нам доступ до назви поточного модуля); таким чином, ми отримали логер, який конфігурували в `settings.py` під назвою ‘`students.signals`’;
- 16ий: сигнал `post_save`³²⁷ серед додаткових параметрів містить `instance`; дана змінна вказує на об’єкт, над яким виконується дія; у нашому випадку це студент, якого редагуємо;
- 17ий: ще одним додатковим параметром даного сигналу є `created`; це булеванівська змінна, яка вказує на те чи поточний об’єкт є новоствореним, а чи вже існує; в обидвох випадках ми логуємо різні повідомлення, щоб відрізняти дію редагування від створення;
- 18ий: з допомогою метода `info` логера записуємо повідомлення про поточну дію, а саме, про створення нового студента; дане повідомлення містить тип події (“`Student added`”), повне ім’я та ID студента, а також решту додаткових параметрів згідно із ‘`verbose`’ форматом;
- 22ий: у випадку, якщо існуючий студент був поредагований, замінюємо стрічку “`added`” на “`updated`”.

Спробуйте тепер перезавантажити ваш сервер і спробувати поредагувати існуючого, а потім додати нового студента.

Ви повинні отримати нові повідомлення у вашій консолі та у файлі з логами.

```
studentsdb.log      [---]  0 L:[ 1+ 0  1/ 4 ] *(0   / 250b) 0073 0x049
INFO 2015-02-16 03:40:12,073 signals: Student updated: Віталій Подоба (ID: 1)
INFO 2015-02-16 03:40:26,371 signals: Student added: demo demo (ID: 19)
INFO 2015-02-16 03:40:31,054 signals: Student updated: Іван Лісовий (ID: 3)
```

studentsdb.log після редагування та додавання студентів

Подібним чином можемо організувати логування при видаленні студента. Єдина відмінність полягає у тому, що в даному випадку ми скористаємося сигналом `post_delete`:

³²⁷<http://djbook.ru/rel1.7/ref/signals.html#post-save>

Обробник видалення студента

```
1 from django.db.models.signals import post_save, post_delete
2
3 @receiver(post_delete, sender=Student)
4 def log_student_deleted_event(sender, **kwargs):
5     """Writes information about deleted student into log file"""
6     logger = logging.getLogger(__name__)
7
8     student = kwargs['instance']
9     logger.info("Student deleted: %s %s (ID: %d)", student.first_name\
10 e,
11         student.last_name, student.id)
```

Думаю вищенаведений код не потребує додаткових пояснень. Ми прив'язались до сигналу `post_delete`, а також змінили стрічку повідомлення на “`deleted`”.

В процесі розробки даних обробників у вас може не вдатись з першого разу реалізувати коректний код. Форми редагування та додавання у нас працюють через технологію AJAX, яка перехоплює усі помилки сервера. Відповідно, це ускладнює процес відлагодження помилок. В такому випадку використовуйте Console панель Firebug плагіна, щоб глянути на відповідь із сервера, або, що у даному випадку буде простіше, відкрийте форму редагування в окремому браузерному вікні (з допомогою пункту контекстного меню на посиланні - Open Link in New Tab). Таким чином, використовуватимете звичайний режим роботи з формою.

Спробуйте тепер видалити існуючого студента і переконайтесь, що дана подія відображається записом в консолі та файлі логування.

Зауважте, що файл логування (`studentsdb.log`) знаходиться в корені репозиторію. Він є генерованим і тому не повинен попадати в репозиторій коду. Тому, вам на домашнє завдання: додати даний файл до списку ігнорованих

файлів. Переконайтесь, що після чергового коміту змін в репозиторії даний файл не попав на віддалений сервер. Ну і, звичайно, не забувайте регулярно комітити останні зміни і пушити їх на віддалений сервер.

В якості домашнього завдання пропоную вам реалізувати логування дій над усіма моделями проекту: студентами, групами та іспитами.

Логуємо помилки при роботі форми контакту

Якщо логування для ведення певного роду статистичних даних є другорядною ціллю, то збереження інформації про помилки роботи програми є значно важливішим завданням логування як такого.

Кожен сервер, програма, веб-аплікація, зазвичай, має вбудований механізм логування помилок. Тому завдання розробника не ламати стандартний механізм і розробляти підходи з додаткового логування помилок при роботі власного кастомного коду.

У нашій аплікації ми можемо бачити в консолі усі критичні помилки з Python кодом та Django шаблонами. Але ми також маємо форму контакту адміністратора. І, якщо відправка листа не спрацьовує, то ми ловимо помилку і відображаємо повідомлення для користувача на формі. Це вже непогано. Але лише уявіть на скільки складно може бути виявити подібну проблему розробнику, а також дізнатись про можливу причину. Такого роду помилку можна виявити, якщо користувач аплікації сконтактує адміністратора, або адміністратор, при тестуванні, сам виявить дану помилку.

Натомість можна реалізувати запис даної помилки в лог файл, а ще краще відправку окремого листа адміністратору при кожній помилці на продакшин сервері.

Ми обмежимось записом помилки в лог файл, а відправка емейла адміністратору залишається вам на домашнє завдання. Для цього в Django уже є все необхідне і вам залишається лише правильно поєднати існуючі компоненти використовуючи інформацію теоретичної частини даної глави.

Різниця між логуванням інформаційної події та помилки полягає у тому, що разом із нашим власним повідомленням, при помилці потрібно включати весь текст помилки, що виникла. В такому випадку користь від такого повідомлення буде значно більшою і дозволить програмісту відслідковувати причину помилки значно швидше.

Python logging модуль уже має необхідний функціонал для логування повного коду помилок. Функція `exception`³²⁸ автоматично включає повний текст помилки в той час, як інші методи приймають опціональний аргумент `exc_info`, щоб включати повний текст про помилку.

Перейдемо одразу до діла і оновимо код функції відправки листа адміністратору. Його насправді небагато, лише кілька стрічок. Тому наводжу лише кілька кусків коду. Якщо важко знайти необхідні місця у вашому модулі, будь-ласка, зверніться до коду, що йде разом із книгою.

Логуємо помилки при відправлення листа

```
1 ...
2 import logging
3 ...
4
5 def contact_admin(request):
6     ...
7     ...
8     try:
9         send_mail(subject, message, from_email, [ADMIN_EMAIL\
10 ])
11     except Exception:
```

³²⁸<https://docs.python.org/2/library/logging.html#logging.Logger.exception>

```
12         message = u'Під час відправки листа виникла непередб\ 
13 ачувана ' \
14                 u'помилка. Спробуйте скористатись даною формою п\ 
15 ізнише.' \
16         logger = logging.getLogger(__name__)
17         logger.exception(message)
18     else:
19         message = u'Повідомлення успішно надіслане!'
20     ...
21     ...
```

Поясню детальніше лише нові рядки коду:

- 2ий рядок: імпортуємо модуль для роботи з логами;
- 16ий: отримуємо логер з іменем поточного модуля; для нього ми уже заготовили логер в модулі `settings.py`; він логуватиме в консоль та файл у форматі ‘verbose’;
- 17ий: використовуючи метод `exception` логера надсилаємо повідомлення про помилку; окрім самого повідомлення, даний метод додасть повний текст помилки, що трапилася в гілці коду `try`.

Щоб потестувати наші оновлення, потрібно спеціально зробити помилку в налаштуваннях SMTP сервера. Інакше лист успішно відправиться і помилка не відбудеться. Найпростіший спосіб зламати відправку - змінити пароль на неправильний.

Спробуйте тепер відіслати повідомлення адміну використовуючи нашу форму контакту. Окрім повідомлення на формі повинні отримати повідомлення та код помилки в консоль та файл логу:

```
ERROR 2015-02-16 04:44:55,223 contact_admin: Під час відправки листа виникла непередбачувана помилка. Стак  
Traceback (most recent call last):  
  File "/data/work/virtualenvs/studentsdb/src/studentsdb/students/views/contact_admin.py", line 64, in contact_admin  
    send_mail(subject, message, from_email, [ADMIN_EMAIL])  
  File "/data/work/virtualenvs/studentsdb/lib/python2.7/site-packages/django/core/mail/__init__.py", line 72, in send  
    return mail.send()  
  File "/data/work/virtualenvs/studentsdb/lib/python2.7/site-packages/django/core/mail/message.py", line 256, in send  
    return self.get_connection(fail_silently).send_messages([self])  
  File "/data/work/virtualenvs/studentsdb/lib/python2.7/site-packages/django/core/mail/backends/smtp.py", line 78, in new_conn_created  
    return self.open()  
  File "/data/work/virtualenvs/studentsdb/lib/python2.7/site-packages/django/core/mail/backends/smtp.py", line 65, in open  
    self.connection.login(self.username, self.password)  
  File "/data/work/buildouts/buildout.python/part/parts/opt/lib/python2.7/smtplib.py", line 615, in login  
    raise SMTPAuthenticationError(code, resp)  
SMTPAuthenticationError: (435, '4.7.8 Error: authentication failed:')
```

Повідомлення про помилку при відправленні емейла

На домашнє завдання пропоную вам переглянути код проекту і запропонувати місця для покращення обробки можливих помилок. Ну і, звичайно, спробувати ці місця покращити згідно нижче наведених двох правил.

...

Тепер ви знаєте, що при виникненні помилки програміст повинен забезпечити два сценарії:

- у зручному форматі повідомити користувача про помилку та запропонувати наступну дію для нього;
- запам'ятати помилку з максимальною кількістю деталей, щоб пізніше програміст міг її відправити.

Якщо ви зможете з легкістю писати ваш код зважаючи на вищенаведені правила, тоді це свідчить про ваш хороший рівень та професіоналізм. Робота з порожніми сторінками (коли ще немає контенту в базі) та помилками програми - це ті важливі деталі, які відрізняють початківця від професіонала.

Домашнє завдання

Як бачите, дана глава видалась більш теоретичною, ніж практичною. Вся справа у тому, що саме логування є досить обширним топіком і дозволяє гнучко

організовувати системи сповіщення у великих проектах. А коли доходить до самого коду, то завдання полягає у кількох тривіальних рядках коду з передачі повідомлення функціям роботи логерів.

Тим не менше, ми з вами розібрались з тим:

- що таке логування і як його використовувати;
- що таке сигнали і яку користь вони нам приносять в Django;
- як правильно обробляти помилки у власному коді.

...

На домашнє завдання вам залишаються усі речі, на які ви натрапили впродовж даної глави:

- журнал подій для усіх решти моделей: студентів, груп та іспитів;
- налаштувати розсилку емейлів адміністраторам сайту про помилки в коді;
- почитати про те, що таке фільтри у Python logging модулі;
- і ще кілька дрібних речей, що ми згадали під час глави.

Також спробуйте розібраться із тим як можна підсвітити ваші лог-повідомлення в консолі різними кольорами взалежності від типу повідомлення. В [даній статті³²⁹](#) є пояснено як це можна зробити на різних операційних системах.

Після того, як ми пройдемо главу по роботі з користувачами в Django, попрошу вас повернутись до цієї глави і реалізвати наступне домашнє завдання: до усіх записів про події над об'єктами в аплікації додати ім'я та ID залогованого користувача, який виконуватиме ту чи іншу дію.

Трохи об'ємніше завдання, щоб потренуватись більше із моделями, базою та в'юшками. Зробіть його в окремій Git гілці, щоб не ламати основної розбницької master бренчі. Якщо все вийде і ви будете задоволені результатом, змерджіть зміни в master. Пропоную переробити логування, щоб кожен запис,

³²⁹<http://plumberjack.blogspot.com.au/2010/12/colorizing-logging-output-in-terminals.html>

замість файлу, йшов у базу даних у вигляді моделі LogEntry. Додайте усі необхідні на ваш розсуд поля і логуйте усі події в базу. Друга частина завдання: розробіть ще одну закладку Події і відображайте там список LogEntry у хронологічному порядку. Застосуйте такі стилі і формат, щоб ви могли пишатись проробленою роботою і отримати естетичне задоволення від візуальної частини.

Створіть власний сигнал і розсылайте його кожного разу, коли відсилається лист адміністратору із форми контакту. Тоді реалізуйте обробник даного сигналу, який логуватиме дію успішної розсылки в лог-файл.

В главі ми з вами ознайомились із сигналами, що мають справу лише з моделями. Тому на домашнє завдання пропоную ознайомитись із ще двома групами подій: обробка запитку та команди manage.py.

Розробіть обробник [сигналу команди migrate³³⁰](#), який після кожного запуску даної команди виводитиме в консоль і лог файл повідомлення про поточну базу даних. Даний сигнал надсилає параметр ‘using’, який дасть вам інформацію про поточну базу даних.

І на завершення, завдання із сигналом запиту [request_started³³¹](#). Напишіть обробник даного сигналу, який рахуватиме кількість запитів. Лічильник можете зберігати в операційній пам’яті (тобто на рівні змінної модуля із сигналами), в базі даних або у файлі. На ваш вибір.

...

До цього часу ми з вами багато Python та HTML файлів “забили” кирилицею. Загалом, це не є на стільки погано, особливо, якщо ви не плануєте далі перекладати ваш проект на кілька інших мов.

Проте в навчальних цілях, у наступній главі розберемось з основами перекладу графічного інтерфейсу веб-аплікації в Django та повністю перекладемо наш проект. Подібний підхід дозволить в майбутньому перекласти даний проект на інші мови, а також дасть вам необхідний мінімум знань в даній області, який вимагається від початківців.

³³⁰<https://docs.djangoproject.com/en/1.7/ref/signals/#post-migrate>

³³¹<https://docs.djangoproject.com/en/1.7/ref/signals/#request-started>

11. Перекладаємо інтерфейс проекту: інтернаціоналізація

Так само як і в попередній главі при розборі ведення журналу подій (логів), переклад та локалізація програмного забезпечення є доволі простою темою з точки зору логіки коду.

В обидвох випадках треба лише володіти необхідним мінімумом теоретичних знань в даній темі, щоб ефективно перекладати ваші програми. Ні алгоритмів, ні складних логічних блоків коду ми не будемо створювати протягом даної глави.

Проте прийдеться ознайомитись із чималою кількістю інформації та набором правил та підходів, які дозволять нам адаптувати нашу веб-аплікацію під потреби користувача, його мову та місцеві формати.

В даній главі ми ознайомимось із:

- поняттям інтернаціоналізації і його застосуванням в Django;
- поняттям локалізації та його застосуванням в Django;
- поняттям часової зони та форматів чисел і дат взалежності від країни;
- методами перекладу та локалізації інтерфейсу в Django: шаблони, Python код та Javascript код.

Дані знання ми застосуємо на практиці:

- правильним чином перекладемо нашу аплікацію на українську мову;
- зробимо так, щоб наша аплікація переключалась між англійською та українською мовами взалежності від налаштування браузера користувача.

Почнемо обов'язково з теорії. Маємо багато інформації для ознайомлення. А уже після теорії перейдемо до практики.

Теорія інтернаціоналізації та локалізації

Основною метою інтернаціоналізації та локалізації є дозволити веб-аплікації надавати свій вміст на різних мовах та форматах пристосованих до користувачів. Тобто, взалежності від налаштувань та місцезнаходження користувача, він може бачити різні мови відвідуючи той самий веб-сайт.

Слова інтернаціоналізація та локалізація доволі часто використовують неправильно. Тому давайте одразу домовимось, що словом “інтернаціоналізація” (або коротко i18n - internationalization) називатимемо процес підготовки програми до перекладу. Процес інтернаціоналізації зазвичай проводять програмісти. Словом “локалізація” (або коротко l10n - localization) називатимемо процес написання тексту для перекладу та локальних форматів. Зазвичай даний процес виконують перекладачі.

Перед тим як перекладач може братись за свою роботу і перекладати текст однієї мови на іншу, програміст повинен пройтись по усіх місцях, що містять текст для користувача (тобто є частиною візуального інтерфейсу програми) і переконатись, що він пристосований для перекладу.

Що взагалі потрібно перекладати, коли мова йде про сайт? На веб-сайті маємо справу з функціональними елементами (різноманітні меню, органи управління сторінками, форми, бічні колонки сторінок, футер і т.д.) та самим вмістом (тіло сторінок, текст блог постів, новини, події і т.д.). Функціональні елементи закладені у веб-сайт програмістом, а вміст сайту - адміністратором та кінцевими користувачами.

Коли говоримо про переклад сайту загалом, то зазвичай маємо на увазі наступний список речей:

- переклад графічного інтерфейсу користувача (тобто функціональні елементи на сторінках);
- вміст веб-сайту, тобто різні версії сторінки взалежності від мови;
- формати дати та чисел взалежності від особливостей країни, де проживає користувач;
- часові зони в датах та часах, знову ж таки, в залежності від місця проживання відвідувача.

Таким чином, окрім самого механізму перекладу функціональних елементів програми та вмісту, кожен веб-сайт, який планує підтримку кількох мов, повинен також мати механізми визначення мови користувача та його місце-знаходження.

Почнемо із тих можливостей, які Django надає для інтернаціоналізації графічного інтерфейсу користувача, тобто функціональних елементів веб-сайту.

Інтернаціоналізація UI елементів

В цій під-секції ми поговоримо про підготовку нашого коду до перекладу. Для цього нам потрібно буде пройтись буквально по усіх файлах, що містять текст, який відображається на сторінках нашої веб-аплікації:

- Python модулі;
- шаблони з HTML кодом;
- Javascript код.

Для кожного із даних типів файлів у фреймворку Django є спеціальний інструментарій інтернаціоналізації. Почнемо із розбору i18n в Python коді.

Python код

Для того, щоб текст в Python коді можна було перекладати на різні мови, його потрібно огорнати у певні функції. Також потрібно, щоб стрічки були англійською мовою. Заведено, що саме англійська є канонічною (тобто основною), а вже інші мови додаються через спеціальні файли перекладів.

Тобто усюди в коді програми ми зустрічаємо лише англійський текст, огорнутий у виклики спеціальних функцій.

Найбільш поширеною функцією, яку ми використовуватимемо буде `ugettext`³³². Данна функція перекладає передану їй стрічку згідно налаштувань проекту та мови користувача (далі в главі ми детальніше поговоримо про визначення поточної мови). Кінцевий результат перекладу є юніковою стрічкою. Дану функцію, як і решту перекладацьких функцій, імпортуємо з пакету “`django.utils.translation`”.

³³²<http://djbook.ru/rel1.7/ref/utils.html#django.utils.translation.ugettext>

Приклад використання ugettext

```
1 from django.utils.translation import ugettext as _
2 from django.http import HttpResponseRedirect
3
4 def demo_view(request):
5     result = _("Hello World.")
6     return HttpResponseRedirect(result)
```

Як бачите, `gettext` ми імпортували під назвою “`_`”. Це, по-перше, зменшить кількість коду для виклику даної функції. А по-друге, дозволить інструментам збору стрічок для перекладу, автоматично організувати для нас файли із перекладами. Дані інструменти пробіжаться по усіх Python модулях і виберуть усі виклики функцій під назвою “`_`”. Також інші розробники одразу розумітимуть, що мають справу із функцією перекладу.

Далі в коді ми викликали функцію “`_`” і передали їй стрічку “Hello World.”. Дано функція одразу перекладає передану стрічку на потрібну мову. Звичайно, для цього ще треба буде організувати файл з перекладами, але про це пізніше.

GNU gettext пакет, який дає нам `gettext` утиліту для обробки файлів із перекладами, є стандартном для реалізації багатомовних програм. Саме тому і більшість i18n функцій в Django містять “`gettext`” у своїй назві.

Виклик функції `gettext` повинен відбуватись лише, коли весь перекладацький Django фреймворк уже є активованим. У попередньому прикладі нашу функцію перекладу ми викликаємо всередині в'юшки. На цей момент Django уже все підготував для перекладів. Проте бувають випадки, коли потрібно перекласти стрічку, яка знаходитьться на рівні імпорту модуля, а не виклику функції чи методу.

Наприклад, усі стрічки, що є значеннями атрибутів класу, запускаються інтерпретатором на момент імпорту Python модуля. В даний момент Django i18n фреймворк може бути ще не активованим.

Приклад коду, де маємо стрічку, яка запускається в момент імпорту модуля

```
1 class Students AppConfig(AppConfig):
2     name = 'students'
3     verbose_name = u"Students Database"
```

Якщо у даному випадку значення атрибути verbose_name огорнути у виклик функції ugettext, то отримаємо наступну помилку:

Занадто ранній виклик ugettext

```
1 django.core.exceptions.AppRegistryNotReady: The translation infrastr\
2 ucture cannot be initialized before the apps registry is ready. Chec\
3 k that you don't make non-lazy gettext calls at import time.
```

Помилка чітко вказує на те, що реєстр Django аплікацій ще не підготований. І, відповідно, структура перекладів також не була поки ініціалізована.

Щоб уникнути подібних проблем в Django також є функції із приставкою “_lazy”. Тобто дані функції роблять все те ж саме, що і їхні основні функції, але **трохи пізніше**³³³. Вони перекладають стрічку не в момент виклику, а в момент використання стрічки. Тобто вони повертають не переклад стрічки на потрібну мову, а спеціальний об'єкт, який в свою чергу повертає потрібний переклад лише при його використанні (наприклад в шаблоні).

Ось як виглядатиме перекладений атрибут verbose_name із вищенаведеного прикладу, з використанням функції ugettext_lazy:

³³³<http://djbook.ru/rel1.7/topics/i18n/translation.html#lazy-translation>

Використання ugettext_lazy

```
1 from django.apps import AppConfig
2 from django.utils.translation import gettext_lazy as _
3
4
5 class StudentsAppConfig(AppConfig):
6     name = 'students'
7     verbose_name = _("Students Database")
8
9     def ready(self):
10         from students import signals
```

Цього разу ми з вами імпортували `gettext_lazy`³³⁴ і скористались нею для значення атрибуту `verbose_name`.

Більшість i18n функцій в Django мають свій “лінівий” (lazy) аналог. Як знати, коли і який використовувати?

По-перше, якщо спочатку користуєтесь не `lazy` варіантом: запускаєте код і якщо ламається з помилкою “`AppRegistryNotReady`” - переключаєте на варіант `_lazy`.

По-друге, код, який запускається інтерпретатором лише раз і використовується під час всього періоду життя сервера (тобто до рестарту), також повинен використовувати `lazy`. Наприклад, описи полів моделей викликаються лише при першому імпорті модулі. Відповідно, різні користувачі із різними мовами бачитимуть лише одну мову - ту яка була активована під час імпорту модуля з моделями. Тому, тут також потрібно `lazy`. На противагу, в коді в'юшок завжди достатньо не `lazy` варіанту, адже в'юшки запускаються кожного разу для кожного запиту. Ось кілька місць, де завжди потрібен варіант `lazy`:

- поля моделей: атрибути `verbose_name`, `help_text`;
- класи моделей: `verbose_name` і `short_description` атрибути.

³³⁴http://djbook.ru/rel1.7/ref/utils.html#django.utils.translation.gettext_lazy

В Python змінна “`_`” є доступна в полі імен по-замовчуванню. Але в Django прийняли рішення завжди явно імпортувати необхідну функцію для перекладу і вже іменувати її під “`_`”. В різних місцях вам знадобляться різні функції, тому використання дефолтної Python змінної “`_`” втрачає сенс.

Також дані функції перекладу підтримують інтерполяцію як і звичайні стрічки в Python:

Передача змінних в стрічку для перекладу

```
1 def demo_view(request, first_name, last_name):
2     result = _('Hello %(first_name)s %(last_name)s!')
3     {'first_name': first_name, 'last_name': last_name}
4     return HttpResponseRedirect(result)
```

Це буває корисно, якщо у двох різних мовах порядок слів є різним, і, відповідно, порядок динамічних компонент у перекладеній стрічці може відрізнятись. Маючи отакі змінні в коді, перекладач отримує свободу самостійно переставити їх у своїх файлах із перекладами.

Стрічки із перекладами також підтримують позиційну інтерполяцію (тобто передаємо не словник, а список із змінними), але рекомендується використовувати лише словник, щоб не забирати від перекладача цю свободу змінювати порядок слів у перекладах.

І на завершення розмови про інтернаціоналізацію в Python коді розглянемо текст із числівниками, де переклад залежить від кількості елементів.

Спеціально для таких випадків в Django є функція `ungettext`³³⁵. Вона дозволяє відображати дві різних версії перекладу: один для однини, інший для множини. Ось приклад із використанням даної функції:

³³⁵<http://djbook.ru/rel1.7/ref/utils.html#django.utils.translation.ungettext>

Приклад використання функції ungettext

```
1 from django.utils.translation import ungettext
2 from django.http import HttpResponseRedirect
3
4 def demo_view(request, count):
5     page = ungettext(
6         'there is %(count)d object',
7         'there are %(count)d objects',
8         count) % {'count': count}
9
10    return HttpResponseRedirect(page)
```

Як бачите, функція `ungettext` приймає 3 аргументи:

- варіант стрічки для однини;
- варіант стрічки для множини;
- змінна-числівник, яка допоможе визначити стрічку, яку використовувати для перекладу.

Ми знаємо, що в українській мові робота з числівниками є значно складніша, ніж в англійській. Нам не обійтись лише двома варіантами. Наприклад, ми можемо мати наступні варіанти:

- “1 яблуко”;
- “2 яблука”;
- “7 яблук”.

Це уже три різних варіанти, взалежності від кількості об'єктів. Але, на щастя, подібних речей немає у нашій веб-аплікації.

Як обходити дане обмеження функції `ungettext`, ми не будемо розглядати в даній главі. Подумайте над можливими обходами даного обмеження, поекспериментуйте і спробуйте винайти свій механізм перекладу.

...

В Django є ще багато інших допоміжних функцій для роботи із стрічками перекладу. Але того, що ми оглянули у даній секції нам вистачить з головою, щоб інтернаціоналізувати увесь Python код нашої веб-аплікації.

i18n в шаблонах

Другим важливим місцем, де часто зустрічаються стрічки, що потребують перекладу, є Django шаблони. Так само як і в Python коді, в Django шаблонах маємо цілий ряд інструментів для інтернаціоналізації тексту.

Щоб скористатись даним інструментарієм треба у кожному із шаблонів завантажувати модуль i18n. Через нього ми отримаємо доступ до усіх наступних тегів.

Тег `trans336` дозволяє перекладати в шаблонах як статичні стрічки, так і динамічні змінні:

Використання тегу trans

```
1  {% load i18n %}  
2  
3  <p>{% trans "Demo String" %}</p>  
4  <p>{% trans demo_variable %}</p>  
5  
6  <p>{% trans "Demo String" as var1 %}</p>
```

На початку даного прикладу ми завантажили модуль i18n. Далі ми скористались тегом `trans`, щоб перекласти статичну стрічку “Demo String”. Зауважте, що ми її огорнули в лапки. Коли ми вдруге використали даний тег, то передали йому змінну під назвою `demo_variable`.

³³⁶<http://djbook.ru/rel1.7/topics/i18n/translation.html#trans-template-tag>

В останньому варіанті використання тегу `trans` ми скористалися ключовим словом “`as`”. Завдяки ньому ми не вставляємо результат роботи тегу в шаблон, а запам'ятуємо перекладену стрічку у змінній під назвою “`var1`”. Це буває корисно, коли потрібно скористатись даною стрічкою у кількох різних місцях далі в шаблоні.

Всередині тег `trans` використовує уже знайому нам Python функцію `gettext`. Під час виконання шаблону дані стрічки будуть перекладені на потрібну мову.

Проте тег `trans` не дозволяє перекладати текст, що містить змінні компоненти. Для цього в Django шаблонах можемо використовувати інший тег: `blocktrans`³³⁷. Даний тег є блочним і містить всередині текст, що потребує перекладу. Даний текст також може містити змінні.

Приклад використання тегу `blocktrans`

```
1  {% load i18n %}  
2  
3  {% trans "бам" as time %}  
4  
5  <p>  
6    {% blocktrans with day=date.day %}I wake up at  
7      {{ time }} o'clock every  
8      {{ day }}{% endblocktrans %}  
9  </p>
```

Спочатку ми, як завжди, завантажили теги модуля `i18n`. Далі присвоїли переклад стрічки “`бам`” у змінну `time`. Потім, всередині тегу параграфа (“`p`”), вставили новий для нас тег `blocktrans`. Важливі моменти, які варто запам'ятати при роботі із даним тегом:

- даний тег є блочним, тобто містить інші теги і текст;
- даний тег може мати опцію `with`, з її допомогою ми маємо можливість визначати динамічні змінні для подальшого використання всередині тегу; у нашому випадку ми визначили змінну `day`, яка бере своє значення з атрибути `day` об'єкта `date`;

³³⁷<http://djbook.ru/rel1.7/topics/i18n/translation.html#std:templatetag-blocktrans>

- змінні всередині тегу `blocktrans` вставляємо як і будь-які інші змінні поза даним тегом, використовуючи подвійні фігурні дужки; ми вставили дві змінні: `time` і `day`.

Також `blocktrans` може містити інший тег: `plural`. Даний тег, в комбінації із `blocktrans`, працює подібним чином як працює Python функція множинного перекладу `gettext`.

Приклад використання тегу `plural`

```
1  {% blocktrans with count apples=tree.apples|length %} 
2  We have the only apple on the tree. 
3  {% plural %} 
4  We have {{ apples }} apples on the tree. 
5  {% endblocktrans %}
```

В цьому випадку до визначення тегу `blocktrans` ми додали ключове слово `count`. Йому присвоєне число - змінна, що одразу йому слідує: `apples`. Дано змінна `count` використовуватиметься тегом `blocktrans`, щоб визначити чи маємо справу з одиною чи множиною, і, відповідно, далі вставляти потрібний текст перекладу. `apples` змінну ми отримали із атрибути `apples` об'єкта `tree` скориставшись фільтром `length`. Даний фільтр повернув кількість об'єктів в масиві `apples`.

Далі, всередині тегу `blocktrans`, ми маємо дві частини тексту. Одна, що передує тегу `plural`, використовується, якщо `count` - одиниця. А все, що йде після тегу `plural`, повертається у випадку, якщо `count` є більше одиниці, тобто маємо справу з множиною.

Ще одну річ, яку варто зауважити щодо тегу `blocktrans`: всередині даного тегу ми не можемо користуватись тегом `url`. Тому будь-які визначення URL адрес повинні відбуватись або до цього тегу, або у визначенні даного тегу. Наприклад:

Визначення URL адрес і тег blocktrans

```
1  {% load i18n %}  
2  
3  {% url 'homepage_view' as home_url %}  
4  
5  {% blocktrans %}  
6      Homepage URL: {{ home_url }}  
7  {% endblocktrans %}
```

Ми підготувались заздалегідь і визначили змінну `home_url` в окремому тезі `url`. Далі, всередині тегу `blocktrans`, просто вставили цю змінну.

Цих двох тегів нам з головою вистачить, щоб перекласти усі шаблони в нашій веб-аплікації.

Django дає нам масу інших тегів, які допомагають [визначати поточну мову³³⁸](#), [переключати мову³³⁹](#), отримувати список доступних мов на сайті та багато інших дій. Проте користуватись ними у даній главі нам не прийеться. Тому їхнє опрацювання залишається вам на домашнє завдання.

i18n в Javascript коді

Подібним чином до того, як ми з вами використовуємо `i18n` функції в Python коді, маємо доступ до функцій перекладу і в Javascript файлах. В Python модулях для того, щоб мати доступ до функцій перекладу, маємо їх спочатку імпортувати. Щоб отримати доступ до подібних функцій на рівні Javascript коду, маємо проробити трохи більше кроків, адже прямого доступу до файлів перекладів мова Javascript, звісно, немає.

Django фреймворк надає нам доступ до в'юшки під назвою “`djongo.views.i18n.javascript_catalog`”. Вона повертає нам Javascript код, що містить масив усіх стрічок, які ми заклали у файли з перекладами (про файли перекладу далі поговоримо детальніше) та набір функцій роботи із перекладом стрічок.

Дану в'юшку потрібно додати до списку URL шаблонів в модулі `urls.py`, а потім включити в якості зовнішнього Javascript файлу на кожну сторінку веб-аплікації.

³³⁸ <http://djbook.ru/rel1.7/topics/i18n/translation.html#other-tags>

³³⁹ <http://djbook.ru/rel1.7/topics/i18n/translation.html#switching-language-in-templates>

Як підключати javascript_catalog в'юшку

```
1 js_info_dict = {  
2     'packages': ('my.package',),  
3 }  
4  
5 urlpatterns = patterns(''  
6     (r'^jsi18n/$', 'django.views.i18n.javascript_catalog', js_info_d\\  
7  ict),  
8 )
```

У вищеннаведеному прикладі ми надали доступ до результату в'юшки javascript_catalog за адресою “/jsi18n”. Ви мабуть уже звернули увагу на змінну js_info_dict. Велика кількість даних в Javascript файлах сповільнює роботу веб-сторінки. Тому варто включати лише ті стрічки з файлів перекладу, які нам потрібні у нашій веб-аплікації. Далі в практичних секціях ми включимо лише стрічки із нашої аплікації students всередині ключа ‘packages’ змінної js_info_dict.

Отже, змінна js_info_dict містить ключ ‘packages’, який містить список заінсталюваних Django аплікацій. В'юшка javascript_catalog пройдеться по даному списку аплікацій і вибере усі стрічки для перекладів. Кожна із даних аплікацій повинна містити папку locale (про неї ми ще поговоримо).

Стрічки для перекладу групуються згідно так званих доменів. Django тримає свої стрічки під доменом django. А усі стрічки, що містяться в Javascript коді, по-замовчуванню приєднуються до домену djangojs. На момент написання книги Django не дозволяв реєструвати стрічки перекладів під власними доменами використовуючи стандартні інструменти скрипта manage.py. Тому усі наші стрічки із файлів перекладів будуть або в домені django, або в djangojs. При розборі файлів з перекладами ми розглянемо, де саме важливо знати про домени стрічок.

Маючи працючу веб-адресу можна підключити в'юшку в якості Javascript файлу на сторінку:

Підключаємо javascript_catalog на веб-сторінку

```
1 <script type="text/javascript"
2         src="{% url 'django.views.i18n.javascript_catalog' %}"></scr\
3 ipt>
```

Також, якщо напряму наберете адресу “<http://localhost:8080/jsi18n>” у вашому веб-переглядачі, то отримаєте щось подібне на оце:

Javascript код, що повертає в'юшка javascript_catalog

Окрім каталогу зі стрічками (`django.catalog`), маємо визначені функції: `gettext`, `ngettext`, `interpolate` і багато інших. Тепер на сторінці є все необхідне, щоб перекладати стрічки тексту, що знаходяться в нашому Javascript коді.

Перекладаємо стрічки

```
1 // gettext: перекладаємо текст для модального вікна:  
2 alert(gettext('Hello world!'));  
3  
4 // ngettext: функція для перекладу однини і множини:  
5 alers(ngettext(  
6     'We have only one apple',  
7     'We have ' + apples.length + ' apples',  
8     apples.length));
```

Бачимо, що обидві функції: gettext і ngettext працюють подібним чином, як і їхні аналоги в мові Python: ugettext і ungettext.

У нас навіть є доступ до функції [interpolate³⁴⁰](#), яка дозволяє вставляти динамічні змінні у стрічку для перекладу.

В'юшка javascript_catalog дає нам значно більше інструментів для роботи із перекладами, але даних двох функцій (gettext та ngettext) нам цілком вистачить для потреб проекту.

Файли локалізації

На даний момент ми повністю розібралися як адаптувати Python код, Django шаблони та Javascript код для того, щоб його можна було далі локалізувати під різні мови.

Але це є лише одна частина роботи. Другою частиною є створення і наповнення файлів, що міститимуть сам текст із перекладами.

Файли перекладу

Файли перекладу зберігаються всередині папки locale в корені проекту або аплікації. Кожна аплікація містить ті переклади, які стосуються саме її функціоналу та графічного інтерфейсу.

Ось як виглядатиме структура папки locale в корені нашої аплікації students:

³⁴⁰<http://djbook.ru/rel1.7/topics/i18n/translation.html#using-the-javascript-translation-catalog>

▼ locale		Today, 2:31 PM	-- Folder
▼ en		Feb 21, 2015 8:21 PM	-- Folder
▼ LC_MESSAGES		Feb 22, 2015 7:08 PM	-- Folder
django.mo		Feb 23, 2015 1:35 PM	4 KB Document
django.po		Feb 23, 2015 1:29 PM	8 KB Document
djangojs.mo		Feb 23, 2015 1:35 PM	4 KB Document
djangojs.po		Feb 22, 2015 7:06 PM	4 KB Document
▼ uk		Feb 21, 2015 8:21 PM	-- Folder
▼ LC_MESSAGES		Feb 22, 2015 7:08 PM	-- Folder
django.mo		Feb 23, 2015 1:35 PM	8 KB Document
django.po		Feb 23, 2015 1:35 PM	12 KB Document
djangojs.mo		Feb 23, 2015 1:35 PM	4 KB Document
		Feb 22, 2015 7:08 PM	4 KB Document

Структура папки locale

В корені папки locale маємо підпапки. Кожна із них відповідає за одну мову перекладу. Ім'я папки співпадає із кодом тої чи іншої мови. [Тут³⁴¹](#) можна ознайомитись із існуючим набором мов та їхніх позначень.

Всередині кожної із даних папок знаходиться ще один рівень підпапок. Усі вони називається LC_MESSAGES. Це загальноприйнята структура для збереження перекладів. А вже в кожній із папок LC_MESSAGES маємо файли, що містять текст перекладу. Відповідно, у нас буде дві мовні папки:

- en: для англійської мови;
- та uk: для української.

Файли перекладу є звичайними текстовими файлами, що представляють текст для перекладу однієї мови. Кожен окремий файл представляє окремий домен. У нашому випадку маємо два домени:

- django: для перекладу усіх стрічок веб-аплікації;
- djangojs: для перекладу лише тих стрічок, що знаходяться в Javascript коді.

Файли перекладу мають розширення “.po”. Також можете зауважити поруч файли із розширенням “.mo”. Це скомпільовані в спеціальний бінарний код файли, що безпосередньо використовуються при перекладі тексту (далі розглянемо як їх створювати).

Ось приклад файла із перекладом на українську мову:

³⁴¹http://en.wikipedia.org/wiki/List_of_ISO_639-1_codes

Приклад файлу з перекладом на українську мову

```
1 #: admin.py:20
2 msgid "Student is a leader of a different group."
3 msgstr "Студент є старостою іншої групи."
4
5 #: apps.py:7
6 msgid "Students Database"
7 msgstr "База Студентів"
8
9 #: models/groups.py:9 models/students.py:44 templates/students/base.\ \
10 html:32
11 #: templates/students/students_add.html:79
12 msgid "Group"
13 msgstr "Група"
14
15 #: models/groups.py:18 templates/students/groups_list.html:20
16 msgid "Leader"
17 msgstr "Староста"
```

Файл поділений на групи, де кожна група представляє собою одну стрічку перекладу і відділена від інших груп одним порожнім рядком. Кожна із стрічок перекладу представлена наступними рядками:

- msgid: ідентифікатор стрічки перекладу; її не можна редагувати вручну; вона береться із канонічної (оригінальної) версії стрічки, яка була використана в коді аплікації; з допомогою даного ідентифікатора унікально прив'язуються переклади доожної із стрічок в коді;
- msgstr: стрічка-переклад на ту мову, для якої призначений поточний ".po" файл; значення даного атрибуту повинно бути огорнутим в подвійні лапки; завдання перекладача полягає у тому, щоб пройтись по усіх даних стрічках і заповнити порожні подвійні лапки стрічками-перекладами;
- також маємо рядок, що починається із стрічки "#: "; тут є список шляхів до файлів, де дана стрічка використовується; як бачите, через двокрапку позначається номер стрічки, де знаходиться дана стрічка у файлі.

Ви мабуть подумали, що це досить клопітка робота проходитьсь по усіх файлах і формувати подібні файли-заготовки для перекладу?

На щастя Django дає розробникам потужний інструмент, який сам генерує усі необхідні файли і навіть структуру папок всередині `locale`. Для цього маємо команду `makemessages`. Наступною командою можемо згенерувати структуру папок для англійської та української мов:

Використання команди makemessages

```
1 $ python manage.py makemessages --locale=uk --locale=en --domain=django
2 ngo
```

Що робить дана команда?

- проходитьсь по усіх папках `locale` в кореняхapplікації та проекту;
- створює усі необхідні підпапки для двох мов всередині `locale`;
- створює файли перекладу в домені `django`; тобто створені файли матимуть назву `django.po`;
- новостворені файли міститимуть заготовки для усіх перекладів знайдених в Python модулях та файлах Django шаблонів; під заготовкою я маю наувазі стрічки готові до заповнення перекладачем.

Ми могли не передавати параметра `domain` у даному випадку, адже він працює по-замовчуванню в даній команді.

Після даної команди можна віддавати подальшу роботу перекладачам, які заповнять порожні параметри `msgstr` потрібними перекладами під обидві мови.

...

Але файли перекладів “`.po`” не використовуються напряму. Їх потрібно спочатку скомпілювати. Компільовані версії файлів перекладів мають розширення “`.mo`” і компілюються з допомогою іншої команди скрипта `manage.py:compilemessages`.

Компілюємо файли з перекладами

```
1 $ python manage.py compilemessages
```

Зауважте, що компільовані файли перекладів є генерованими файлами, а відповідно і не повинні ніколи попадати у репозиторій коду. На продакшин системах, як і на розробницьких, кожного разу повинна запускатись команда `compilemessages` після оновлення перекладів. Як варіант, також можна інтегрувати компіляцію перекладів у процес деплойменту. До речі, ми уже давніше внесли файли із розширенням “.mo” у список ігнорованих файлів з допомогою файлу “.gitignore”.

Усі стрічки, що знаходяться в Javascript коді будуть проігноровані вищезгаденою командою `makemessages`. Ми збириали стрічки, що належали до домену `django`, який є дефолтним в Django. А от стрічки у Javascript коді, як ми вже з вами з’ясували, належать до домену `djangojs`. Відповідно, для оновлення файлів перекладів `djangojs`.ро потрібно викликати наступну команду:

Збираємо переклади для Javascript стрічок

```
1 $ python manage.py makemessages -d djangojs -l uk -l en
```

Зверніть увагу, що в останньому прикладі ми скористались скороченими версіями опцій для передачі домена та списку мов для перекладів: “`-d`” та “`-l`”.

Також хочу зауважити, що кодування усіх файлів з перекладами повинне бути UTF-8. Тому переконайтесь, що ваш редактор коду відповідно налаштований.

У випадку, якщо вам потрібно одразу оновляти файли перекладів усіх мов, тоді можете користуватись опцією “-а”, яка пройде по усіх мовах і оновить необхідні файли.

gettext утиліта

Якщо після запуску команди makemessages ви отримали помилку, або ваші щойно згенеровані файли із перекладами є порожніми, тоді, швидше за все, бракує бібліотек у вашій операційній системі.

Для збору та організації файлів перекладів команда makemessages використовує утиліти із набору [GNU gettext³⁴²](#): xgettext, msgfmt, msgmerge, msguniq.

Тут знайдете інструкції щодо інсталяції та налаштування утиліт gettext на трьох популярних операційних системах:

- [Linux Ubuntu³⁴³](#);
- [Windows³⁴⁴](#);
- [Mac OS³⁴⁵](#); інсталюєте використовуючи один із портів: port або brew.

На Лінуксових системах дані утиліти зазвичай є одразу доступними. А Windows та Mac OS користувачі повинні будуть їх доставити.

...

На цьому секція із перекладу графічного інтерфейсу користувача завершена. Ми тепер знаємо як адаптувати наш код, а також як готовити структуру папок та файлів із перекладами для роботи перекладача.

Переклад контенту

До цього моменти ми з вами розбиралі як перекладається та частина аплікації, яка відповідає за функціонал та графічний інтерфейс користувача. Цього нам буде достатньо, щоб виконати усі практичні завдання даної глави.

³⁴²<https://www.gnu.org/software/gettext/>

³⁴³<http://packages.ubuntu.com/trusty/gettext>

³⁴⁴<http://djbook.ru/rel1.7/topics/i18n/translation.html#gettext-on-windows>

³⁴⁵<https://gist.github.com/mbillard/1647940>

Але для повноти картини щодо перекладу та локалізації веб-сайту також коротко розглянемо другу частину перекладацької роботи: контент, або ще як його називають вміст сайту.

Велика частина існуючих веб-сайтів в мережі інтернет обслуговує користувачів одразу із кількох різних країн, що володіють різними мовами. Щоб цього досягнути, потрібно перекладати не лише функціональну частину сайту, але й дублювати текст кожної сторінки, новини, події чи блог поста на кожну із мов.

При цьому різні мови сайту можна представляти користувачеві по-різному:

- на різних доменах, наприклад: google.com, google.com.ua, google.co.uk;
- на різних під-доменах, наприклад: en.mysite.com, uk.mysite.com;
- на різних під-шляхах в межах одного домену, наприклад: example.com/en, example.com/uk;
- і як варіант: одна і та ж адреса може показувати різні мови сторінки взалежності від активної мови.

Переклад вмісту веб-сайту також складається із двох частин. Перша частина - це підготовка інструментарію для перекладача, з допомогою якого він зможе легко готувати копії однієї і тієї ж сторінки на різних мовах. Дану частину виконує розробник.

Кожна із популярних веб CMS йде із власним подібним інструментом для перекладу вмісту на сайті, або радить, який із зовнішніх існуючих інструментів (це може бути аплікація, плагін, додаток) використовувати у себе на сайті.

Django не є CMS системою, а фреймворком, а тому і варіанти для перекладу саме вмісту вашого веб-сайту залежатимуть цілком від того, як ви реалізуете вашу аплікацію. Наприклад, ви можете набивати ваші моделі полями-дублікатами для перекладів на кожну нову мову. Або ви можете організувати переклад моделей у вигляді окремих класів моделей - новий клас тої ж моделі для нової мови.

Переклад значень полів моделей є доволі поширеним завданням, тому після деякого часу експериментів в Django спільноті з'явилися кілька аплікацій, які дозволяють додавати переклади у ваш сайт без додаткового кодування.

Ось [список³⁴⁶](#) усіх існуючих рішень для перекладів контенту в Django. Коли вибираєте, яке підходить саме вам, керуйтесь вимогами проекту. Мінімальний список критеріїв:

- підтримує версію Python, на якому працює ваш проект;
- підтримує версію Django вашого проекту;
- підтримує команду `migrate`; в таблиці позначений даний критерій як `South support`;
- кількість людей, які підписані на даний пакет в репозиторії коду: `Repo Wathers`;
- чи наявна документація;
- чи є тести;
- інтеграція із дефолтною Django адміністративною частиною.

Якщо усі обов'язкові критерії задовольняють кілька різних аплікацій, тоді обираєте найпопулярнішу. За інших рівних умов, рекомендую обирати перший інструмент у списку: (`django-hvad`) [<https://github.com/KristianOellegaard/django-hvad>].

Взагалі, через специфіку Django ORM завдання уніфікації перекладу моделей (тобто даних в базі) є доволі складним. Саме тому на даний момент ще поки не існує однієї аплікації, яка б підходила на усі випадки. Відповідно, з досвідом ви почнете розуміти різницю між тією чи іншою аплікацією, її сильні та слабкі сторони саме для вашого випадку. А поки, перед тим як інсталювати та використовувати одну із аплікацій перекладу даних в моделях - робіть регулярні бекапи бази даних ;-)

i18n в URL шаблонах

Якщо у вашому проекті ви вирішите обслуговувати різні мови сторінок на одному домені з різними під-шляхами, тоді вам може стати у пригоді властивість самого Django для перекладів URL шаблонів.

Django надає два способи інтернаціоналізації URL шаблонів:

³⁴⁶<https://www.djangoproject.com/grids/g/model-translation/>

- динамічно додавати мовні префікси до ваших URL адрес з допомогою `i18n_patterns`³⁴⁷ функції;
- або перекладати URL шаблони використовуючи функцію `gettext_lazy`³⁴⁸ та файли перекладів.

В обидвох випадках потрібно додати мідльвару `django.middleware.locale.LocaleMiddleware`³ до списку `MIDDLEWARE_CLASSES`³⁵⁰ в `settings.py` модулі.

У варіанті з мовними префіксами і функцією `i18n_patterns`, використовуєте дану функцію замість функції `patterns`:

Приклад використання функції `i18n_patterns` в `urls.py`

```

1 from django.conf.urls import include, url
2 from django.conf.urls.i18n import i18n_patterns
3
4 from blog.urls import urlpatterns as blog_patterns
5 from events.urls import urlpatterns as events_patterns
6
7
8 urlpatterns = i18n_patterns(
9     url(r'^$', 'path.to.homepage.view', name='homepage'),
10    url(r'^blog/$', include(blog_patterns), namespace='blog'),
11    url(r'^events/$', include(events_patterns, namespace='events')),
12 )

```

У вищезгаданому куску коду бачимо два нових підходи:

- ми скористались функцією `include`³⁵¹, яка дозволяє вкладати цілу структуру веб-адрес під один шлях; у нашому випадку ми включили набір шаблонів у секцію сайту /blog, а також /events; дана функція `include`

³⁴⁷http://djbook.ru/rel1.7/topics/i18n/translation.html#django.conf.urls.i18n.i18n_patterns

³⁴⁸http://djbook.ru/rel1.7/ref/utils.html#django.utils.translation.gettext_lazy

³⁴⁹<http://djbook.ru/rel1.7/ref/middleware.html#django.middleware.locale.LocaleMiddleware>

³⁵⁰http://djbook.ru/rel1.7/ref/settings.html#std:setting-MIDDLEWARE_CLASSES

³⁵¹<http://djbook.ru/rel1.7/ref/urls.html#django.conf.urls.include>

дозволяє кожній із аплікацій визначати свій набір URL шаблонів, а вже те, як вони використовуватимуться в тому чи іншому проекті, залежатиме від налаштувань urls.py в корені проекту;

- замість patterns ми скористались функцією i18n_patterns, яка надає додаткову динаміку нашим веб-адресам; до кожного із шаблонів дана функція додасть префікс - URL компоненту тієї мови, яка зараз активована на сайті; таким чином кожного разу, коли ви використовуватимете url тег або Python функцію reverse, щоб отримати URL адресу до секції блогу, отримуватимете “/en/blog/” адреси (у випадку, якщо поточна активована мова є англійська).

Використовуючи другий спосіб перекладу URL адрес, потрібно перекладати необхідні URL шаблони з допомогою функції ugettext_lazy. Давайте одразу до прикладу:

Перекладаємо шаблони URL adres

```
1 from django.conf.urls import include, url
2 from django.conf.urls.i18n import i18n_patterns
3 from django.utils.translation import ugettext_lazy as _
4
5 from events.urls import urlpatterns as events_patterns
6
7
8 urlpatterns = i18n_patterns(
9     url(r'^$', 'path.to.homepage.view', name='homepage'),
10    url(_(r'^contact/$'), 'path.to.contact.view', name='contact'),
11    url(_(r'^events/')), include(events_patterns, namespace='events')\
12 ),
13 )
```

Як бачите, ми поєднали обидва підходи і отримали адреси, які одночасно містять перекладені компоненти та містять мовний префікс. Приклад адреси сторінки “contact” при активованій німецькій мові буде: “/de/kontakt”. Англійське слово contact перекладено на німецьку з допомогою файлів перекладів, а префікс “/de/” додала функція i18n_patterns.

Рекомендується поєднувати цих два підходи для утворення веб-адрес багатомовних сторінок на сайті.

Також, якщо ви вирішили обслуговувати різні мови на різних під-шляхах вашого веб-сайту і хочете уникнути модифікації вашого urls.py модуля, тоді аплікація [django-localeurl³⁵²](#) вам стане у пригоді. Данна аплікація зробить більшість роботи за вас.

Ми не будемо детальніше оглядати тему перекладу вмісту сайту. Практичне застосування даної теми також залишається поза межами даної книги. Щоб трохи попрактикуватись, наприкінці глави буде описане відповідне домашнє завдання.

Локалізація дат, часів та чисел

В цій секції розглянемо поняття часової зони, а також розберемось, що таке форматування дати та чисел і як воно залежить від місцезнаходження відвідувача сайту.

Почнемо із часової зони. Мабуть найоб'ємнішої теми:

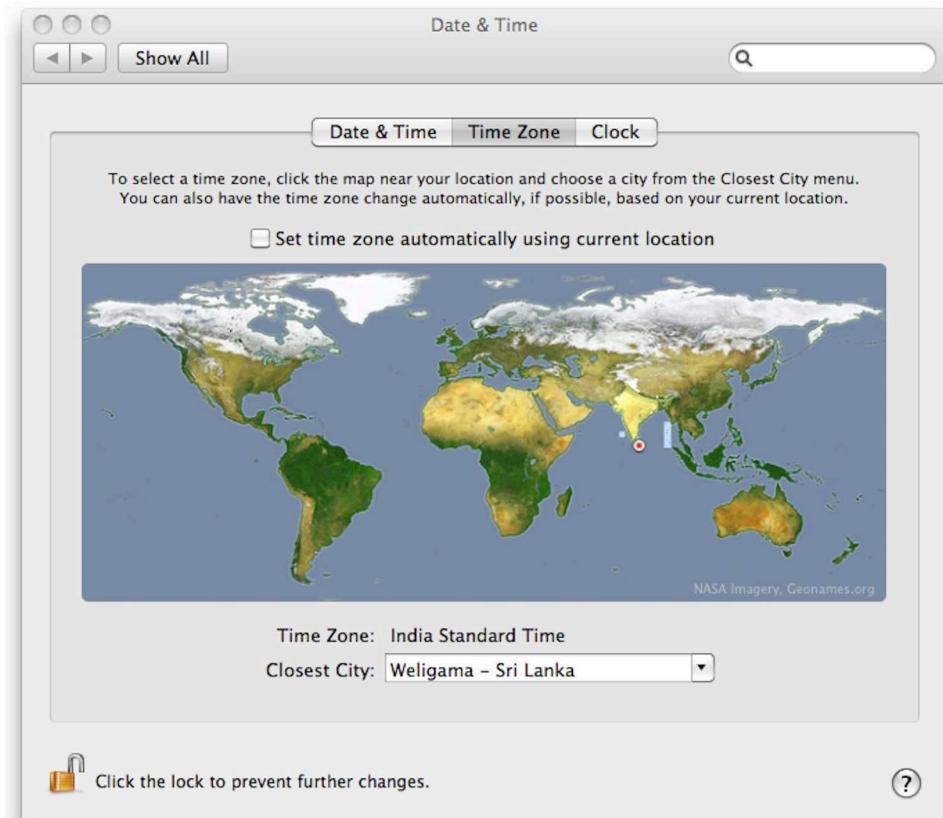
Часові зони

Ще до кінця 19 століття люди встановлювали свій локальний час орієнтуючись на схід та захід сонця. На планеті Земля обидві події відбуваються у різний час в різних місцях через те, що планета крутиться навколо своєї осі. Високошвидкісного транспорту в той час не існувало. Комунікації на великих відстанях не існувало також.

Але з розвитком обидвох, прийшла потреба уніфікувати системи для встановлення часу в різних куточках нашої планети. В 1884 році за точку відліку світового часу прийняли меридіан Грінвіча. Від нього в обидві сторони віртуально розділили глобус на сегменти у 15 градусів довготи, який відповідав одній годині в сонячній системі.

³⁵²<https://pypi.python.org/pypi/django-localeurl>

На практиці ж дані сегменти порушуються межами країн. В даний час навіть існують часові пояси, що відрізняються на пів-годинний зсув від нульового меридіана.



Часові зони світу, налаштування в Mac OS

Існує багато систем вимірювання часу. Ми використовуватимемо UTC³⁵³ - всесвітній координований час. Відповідно, часова зона по Гринвічу позначатиметься як UTC-0, або просто UTC, або UTC+0. Тобто нульовий зсув відносно меридіана Гринвіча.

Часовий пояс України³⁵⁴ - UTC+2. Проте, в нас, як і в частини інших країн світу, діє літній час. Це означає, що часовий пояс влітку переходить у UTC+3.

³⁵³ <https://ru.wikipedia.org/wiki/UTC%C2%B100:00>

³⁵⁴ <http://www.timeanddate.com/worldclock/ukraine/kyiv>

Також замість UTC часто вживають форму GMT - Greenwich Mean Time - Середній Час по Грінвічу.

...

Тепер давайте трохи повернемось до веб-сайтів.

Якщо ваш веб-сайт призначений для українських відвідувачів і сам хоститься в українського провайдера, тоді додаткових рухів з вашої сторони не потрібно.

Якщо ваш веб-сайт розміщений на сервері в одній країні, а обслуговує користувачів іншої країни (або для багатьох різних країн), тоді в якій часовій зоні відображати дати на сайті?

Є два варіанти:

- зберігаємо і показуємо дати у часовій зоні сервера, де крутиться ваша веб-аплікація; це найпростіший підхід і, зазвичай, також не вимагає додаткових налаштувань чи коду; проте даний підхід може бути не таким зручним для ваших користувачів; уявіть, що у фейсбуці ви бачите пости ваших друзів не у вигляді “2 години тому”, а “14:00 UTC+2”, або щось подібне; досить важко зрозуміти як давно запостили новину без додаткових обрахунків в голові;
- зберігати час в універсальній зоні (UTC), а показувати на сторінках сайту кожному відвідувачу адаптований до його часової зони перебування; так роблять більшість соціальних мереж; погодьтеся - зручно; звичайно, і реалізувати такий підхід значно складніше.

Для другого варіанту Django і Python надають цілий ряд інструментів, які допомагають:

- визначати часову зону відвідувача;
- конвертувати дату і час між різними зонами;

- зберігати дату і час в UTC зоні.

Часові зони в Python

Загалом, робота з датами і часовими зонами в мові Python не є найсильнішою стороною даної мови. В ході історії розвитку мови розробники прийняли кілька невдалих рішень, потім невдалих спроб виправити ситуацію і в кінцевому результаті маємо схему роботи далеку від інтуїтивної.

Python `datetime.datetime` об'єкти мають атрибут `tzinfo` призначений для зберігання інформація про часову зону. Інформація про часову зону представлена об'єктом класу `datetime.tzinfo`. Якщо інформація присутня, тоді об'єкт дати називаємо `aware` (тобто, обізнаний про часову зону). Інакше, називаємо таку дату `naive` (з англ. наївний, простий), тобто дата без інформації про часову зону.

Простий і один з найкращих підходів для роботи із датами - це зберігати усі дати без інформації про часову зону, але конвертувати ці дати в UTC зону, тобто початок відліку часу. А вже, коли потрібно представляти дату чи час на користувачькому інтерфейсі в певній часовій зоні - на ходу конвертувати зі UTC зони.

Це подібно до того, як рекомендується працювати із кодуванням тексту: працюємо в аплікації виключно із юнікодом, а повертаємо клієнту в UTF-8 кодуванні.

Через те, що робота із часовими зонами в Python доволі не проста, рекомендую використовувати додаткову бібліотеку `pytz`³⁵⁵. Вона значно спростить ваше життя, якщо потрібно багато працювати із датами і часами в різних часових зонах.

В [даній статті](#)³⁵⁶ зможете детальніше розібратись із нюансами роботи з часовими зонами в мові Python.

³⁵⁵<https://pypi.python.org/pypi/pytz>

³⁵⁶<http://habrahabr.ru/company/mailru/blog/242615/>

Часові зони в Django

Django дає нам ще кілька покращень.

По-перше, просто увімкнувши підтримку часових зон, фреймворк:

- почне зберігати в базі усі об'єкти дати і часу в UTC зоні;
- до усіх об'єктів `datetime` прикріплятиме інформацію про часову зону;
- і переведитиме їх в часову зону користувача на графічному інтерфейсі (шаблони, форми).

Якщо ж підтримка часових зон виключена, тоді усі об'єкти дат і часів, відповідно, не мають приєднаної інформації `tzinfo`.

По-друге, Django дає набір функцій, щоб працювати із датами та часовими зонами:

- `is_aware357`: перевіряє чи об'єкт дати містить часову зону;
- `is_naive358`: перевіряє чи об'єкт дати є датою без часової зони;
- `timezone.now359`: повертає поточну дату і час із дефолтною часовою зоною Django проекту;

Також є набір інструментів для визначення поточної часової зони відвідувача, але про це детальніше поговоримо в наступній секції.

Налаштування роботи із часовими зонами виконується через налаштування проекту:

`settings.py` модуль всередині проекту

```
1 USE_TZ = True
2 TIME_ZONE = 'UTC'
```

- `USE_TZ360`: включаємо часові зони в проекті;

³⁵⁷ http://djbook.ru/rel1.7/ref/utils.html#django.utils.timezone.is_aware

³⁵⁸ http://djbook.ru/rel1.7/ref/utils.html#django.utils.timezone.is_naive

³⁵⁹ <http://djbook.ru/rel1.7/ref/utils.html#django.utils.timezone.now>

³⁶⁰ http://djbook.ru/rel1.7/ref/settings.html#std:setting-USE_TZ

- **TIME_ZONE³⁶¹**: дефолтна часова зона аплікації; фреймворк зберігатиме дати проекту в базі у даній часовій зоні; рекомендоване значення UTC; якщо USE_TZ включене, тоді дана часова зона також буде використовуватись для відображення дат користувачам.

...

Якщо підтримка часових зон включена, тоді ваш код повинен використовувати функцію now Django модуля **django.utils.timezone³⁶²** для створення поточного часу:

Поточна дата і час, якщо підтримка часових зон включена

```
1 from django.utils import timezone
2
3 now = timezone.now()
```

Так ми отримаємо час і дату, що є прив'язані до часової зони, активованої в проекті. Також, як уже зазначено вище, варто використовувати пакет pytz, щоб полегшити свою роботу із датами в часових зонах.

В протилежному випадку достатньо використовувати Python datetime модуль:

Поточна дата і час, якщо підтримка часових зон виключена

```
1 import datetime
2
3 now = datetime.datetime.now()
```

...

Також Django дає цілий набір тегів для роботи із датами в шаблонах. Усі ці теги потрібно додатково завантажити на рівні кожного шаблона із модуля tz.

Тег **localtime³⁶³** дозволяє вказати об'єкту дати і часу, що міститься в ньому, чи конвертуватись до поточної зони:

³⁶¹<http://djbook.ru/rel1.7/ref/settings.html#time-zone>

³⁶²<http://djbook.ru/rel1.7/ref/utils.html#module-django.utils.timezone>

³⁶³<http://djbook.ru/rel1.7/topics/i18n/timezones.html#localtime>

Використання тегу localtime

```
1  {% load tz %}  
2  
3  {% localtime on %}  
4      {{ mydate }}  
5  {% endlocaltime %}  
6  
7  {% localtime off %}  
8      {{ mydate }}  
9  {% endlocaltime %}
```

Як видно із вищезгаданого прикладу, тег localtime отримує параметр on або off, щоб відповідно задіяти або деактивувати конвертацію дати в локальний час (поточну часову зону).

Щоб отримати кращий контроль над часовою зоною для об'єкту дати, маємо тег timezone. Він дозволяє активувати ту чи іншу часову зону для коду, що є всередині нього. Доволі гнучко і динамічно:

Приклад використання тегу timezone

```
1  {% load tz %}  
2  
3  {% timezone "Ukraine/Kiev" %}  
4      Ukraine time: {{ dateobj }}  
5  {% endtimezone %}  
6  
7  {% timezone None %}  
8      Default server time: {{ dateobj }}  
9  {% endtimezone %}
```

І третім важливим тегом є тег `get_current_timezone`³⁶⁴. Він повертає поточну часову зону. Це може бути динамічно визначена зона відвідувача або дефолтна зона налаштована в модулі `settings.py`

³⁶⁴<http://djbook.ru/rel1.7/topics/i18n/timezones.html#get-current-timezone>

...

На цьому завершуємо коротенький екскурс в часові зони і способи роботи з ними в Django. Насправді тема доволі обширна і складна в реалізації. Лише реальні практичні проекти дадуть можливість сповна розібратись із даною темою і набити руку.

Далі переходимо до форматування дат, часу та чисел.

Формат дати, часу та чисел

Ви мабуть зустрічались із тим, що на різних закордонних сайтах одні і ті ж числа записують по різному. Наприклад:

- в Канаді: 4 294 967 295,000;
- в Італії: 4.294.967.295,000;
- у Великобританії: 4,294,967,295.00.

В одних використовується крапка, в інших - кома, ще в інших - пробіл.

Те ж саме стосується і формату дат та часів. В більшості країн відображення дати і часу *відрізняється*³⁶⁵.

Відповідно, якщо на ваш сайт приходять користувачі із різних країн, їм було б зручніше бачити числа та дати у їхньому звичному форматі. Подібно до того як приводяться часові зони для кожної окремої країни.

Для локалізації форматів чисел та дат Django фреймворк також дає нам ряд інструментів. А щоб включити підтримку локалізації в налаштуваннях проекту (модуль `settings.py`) потрібно встановити змінну `USE_L10N`³⁶⁶ в `True`. При включеній локалізації усі дати та числа, що потраплятимуть в Django шаблони, автоматично відображатимуться в локалі користувача.

Серед інструментів локалізації Django надає:

- атрибут `localize` для локалізації полів у формах;

³⁶⁵http://en.wikipedia.org/wiki/Date_format_by_country

³⁶⁶http://djbook.ru/rel1.6/ref/settings.html#std:setting-USE_L10N

- теги для форматування дат та чисел в шаблонах.

Кожне тип поля форми має аргумент `localize367`. Якщо він встановлений у `True`, тоді значення поля на формі буде представлене у локалізованому для користувача форматі.

Приклад використання аргументу `localize` в полі `DecimalField`

```
1 class DemoForm(forms.Form):
2     height = forms.DecimalField(max_digits=3, decimal_places=2, local\
3         ize=True)
```

У вищеприведеному прикладі ми визначили клас форми з єдиним полем `height` (з англ. висота). Дробове число, в якості значення даного поля, буде локалізоване при виводі форми на інтерфейс користувача завдяки переданому аргументу `localize`.

Також маємо тег `localize368` в Django шаблонах, щоб керувати форматами дат та чисел. Щоб отримати доступ до даного тегу, потрібно завантажити модуль `l10n`. Ось приклад використання тегу:

Приклад використання тегу `localize`

```
1 {% load l10n %} 
2 
3 {% localize on %}
4     {{ mynumber }} 
5     {{ mydate }} 
6 {% endlocalize %}
```

Тег `localize` дозволяє включати та виключати процес локалізації дат та чисел на рівні елементів, що знаходяться всередині нього. Якщо ми не використовуємо даний тег, тоді локалізація застосовується згідно налаштувань глобального параметра `USE_L10N`.

³⁶⁷ <http://djbook.ru/rel1.7/ref/forms/fields.html#localize>

³⁶⁸ <http://djbook.ru/rel1.6/topics/i18n/formatting.html#localize>

В прикладі вище ми активували локалізацію для змінної `mynumber`. Якщо б нам потрібно було тимчасово деактивувати локалізацію, тоді потрібно скористатись аргументом `off` для тега `localize`.

Таким чином, весь вміст тега `localize` буде локалізуватись (або ні) згідно налаштувань його налаштувань. Якщо ж нам потрібно включити або відключити локалізацію для єдиного значення чи змінної, тоді достатньо скористатись фільтрами:

- `localize369`: активуємо локалізацію для значення;
- `unlocalize370`: деактивуємо локалізацію для даного значення.

Для використання даних фільтрів також потрібно завантажити модуль `l10n`:

Приклад застосування фільтрів локалізації

```
1  {% load l10n %}  
2  
3  {{ mydate|localize }}  
4  
5  {{ mynumber|unlocalize }}
```

З допомогою даних фільтрів можемо керувати локалізацією на рівні окремих змінних.

...

В попередніх версіях Django фреймворк містив набір так званих локалей для різних країн, мов та територій. Згодом цю колекцію винесли в окрему аплікацію `django-localflavor371`.

³⁶⁹<http://djbook.ru/rel1.6/topics/i18n/formatting.html#std:templatefilter-localize>

³⁷⁰<http://djbook.ru/rel1.6/topics/i18n/formatting.html#unlocalize>

³⁷¹<https://django-localflavor.readthedocs.org/en/latest/>

Локаль - це набір сталих форматів, перекладів та дат прийнятих в тій чи іншій країні або території.

Дана аплікація містить:

- набір додаткових полів для форм специфічних для різних країн;
- колекції адміністративних одиниць поділу країн;
- універсальні утиліти при роботі із локалізацією інтерфейсу.

Вбудованого Django функціоналу разом із даною аплікацією вистачить для покриття 90% випадків, які вам можуть знадобитись у ваших проектах. Решту випадків можна перекрити за допомогою реалізації [власних форматів](#)³⁷². Але це залишається за межами даної книги.

В процес локалізації також може входити переклад одиниць виміру, валют, розмірів одягу та взуття. Останнім часом стало популярним отримувати локацію користувача на сторінках веб-сайту. Особливо серед користувачів мобільних телефонів. Все це також використовується, щоб запропонувати відвідувачу контент відповідно до його місцезнаходження.

Як Django знає про мову та часову зону користувача?

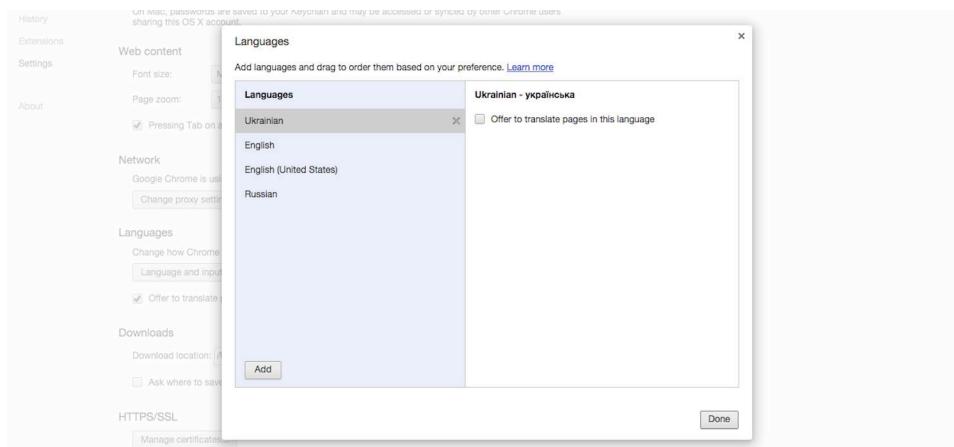
Ми коротенько розглянули усі необхідні концепції щодо локалізації та інтернаціоналізація як загалом, так і в контексті Django фреймворку. Єдина річ, яку залишилось вияснити: Як Django знає про поточну мову користувача, його часову зону та місцезнаходження?

Дана інформація береться з різних джерел.

³⁷²<http://djbook.ru/rel1.6/topics/i18n/formatting.html#creating-custom-format-files>

Мова користувача

Отримати мову користувача є найпростішим завданням. HTTP протокол має стандартизований заголовок Accept-Language, який містить мову користувача. Мову користувач сам вибирає у своєму веб-переглядачі. Кожен сучасний браузер дозволяє вибрать список мов і даний список відсилається на сервер при кожному запиті. Порядок мов має значення - це пріоритет, в якому відвідувач бажає отримувати переклад веб-сайту:



Налаштування браузера Chrome

Приблизно так виглядає мій заголовок мовних налаштувань у браузері:

Заголовок Accept-Language

1 Accept-Language:uk,en;q=0.8,en-US;q=0.6,ru;q=0.4

Даний заголовок містить список мов і їх важливість. Параметр q вказує на пріоритет мови як частину від одиниці. Таким чином, із заголовку вище бачимо, що українська мова є моєю пріоритетною мовою, але я також можу читати англійською та російською, якщо веб-сайт надає такий переклад.

Django фреймворк бере до уваги даний заголовок і активує найпріоритетнішу мову для користувача. Звісно, якщо вона доступна серед існуючих мов веб-сайту. Якщо ж даний заголовок, з якихось причин, відсутній, тоді веб-аплікація відображатиметься у дефолтній мові.

Частина веб-сайтів дозволяє переключатись між мовами з допомогою візуальних органів управління: мовних флагків, випадаючого меню з мовами. З їх допомогою, навіть анонімний користувач може переглядати веб-сайт на різних мовах.

Альтернативно, частина веб-аплікацій реалізує форму налаштувань користувача, де зареєстрований користувач може вибирати мову відображення на веб-сайті. Таким чином, залоговані користувачі використовують веб-аплікацію на своїй рідній мові.

В обидвох останніх випадках, такий вибір користувача має перевагу над заголовком Accept-Language.

Часова зона і локаль користувача

Нажаль, в специфікації протоколу HTTP не існує заголовка, який міг би передавати від клієнта на сервер часову зону користувача.

Більшість веб-сайтів вирішують дану проблему надаючи залогованому користувачу вибирати свою часову зону у своїх налаштуваннях.

Альтернативно, тепер браузери підтримують визначення локації користувача. Якщо відвідувач сайту дозволяє надати свою геолокацію веб-сайту, тоді він може отримати спеціалізований вміст, мову і часову зону на сторінках даного ресурсу.

Для анонімів веб-сайти, зазвичай, показують контент в часовій зоні, де проживає більшість відвідувачів. Або в UTC зоні.

Django веб-фреймворк надає ряд інструментів для організації інтерфейсу вибору часової зони на користувацьких формах. Його ми розглядати не будемо. Важливо, щоб ви знали про таку можливість іскористались нею при наступній потребі.

...

На цьому ми завершуємо нашу теоретичну частину і переходимо до практики, де перекладемо сторінки веб-аплікації на українську мову. Тепер ми володіємо необхідним мінімумом знань, щоб справитись із даною задачею.

Готуємось до інтернаціоналізації

Щоб зайнятись інтернаціоналізацією шаблонів, Python та Javascript коду, потрібно підготувати інструменти, структуру папок та налаштування проекту.

Версія Django фреймворку

У версіях Django до 1.7.2 існувала одна доволі серйозна проблема із перекладами, яка не дозволяла користуватися командами скрипта `manage.py` для збору та генерація файлів перекладів.

Щоб дізнатись версію вашої Django інсталяції, скористайтесь опцією `version` скрипта `manage.py`:

Як отримати версію Django

```
1 # будучи в корені репозиторію
2 # і активувавши віртуальне середовище
3 $ python manage.py --version
4 1.7.2
```

Якщо у вас версія Django старіша, ніж 1.7.2, обов'язково оновіть її:

Оновлюємо версію Django

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb
2 # активуйте ваше віртуальне середовище перед даною командою
3 # переконайтесь, що команда закінчилася успішно
4 $ pip install Django==1.7.2
```

На момент написання книги, остання стабільна версія Django була 1.7.7. Вона, а також усі версії в серії 1.7.x повинні працювати з кодом даного проекту. Книга також буде підтримуватись для усіх наступних версій Django.

Необхідні інструменти перекладу

Ми вже з вами обговорили, які утиліти та бібліотеки потрібні для створення перекладів. Тому, будь-ласка, поверніться до попередньої секції та встановіть все необхідне, якщо ви цього ще не зробили.

Без даних інструментів інтернаціоналізація та локалізація веб-аплікації буде неможливою. Лише після цього рухайтесь далі.

Налаштування проекту

Час активувати локалізацію та інтернаціоналізацію на рівні Django проекту. Для цього відкриємо налаштування проекту в модулі `settings.py` і переконаємось, що необхідні змінні встановлені:

Налаштування l10n та i18n проекту

```
1 USE_I18N = True
2 LANGUAGE_CODE = 'uk'
3 USE_L10N = True
4 USE_TZ = True
5 TIME_ZONE = 'UTC'
```

Ми уже розібрались із даними змінними в теоретичній частині, тому давайте тут лише коротенько згадаємо:

- USE_I18N: активуємо інтернаціоналізаційну систему Django
- LANGUAGE_CODE: код мови, яка буде використовуватись в проекті по замовчуванню; цю змінну можна перекривати іншими налаштуваннями; один із прикладів розглянемо наприкінці глави; при виключенні USE_I18N дана змінна LANGUAGE_CODE немає жодної сили;
- USE_L10N: активуємо локалізацію чисел та дат; тобто кожен користувач бачитиме числа та дати у форматі прийнятому у його країні;
- USE_TZ: вказуємо Django зберігати і використовувати дати, які мають інформацію про часову зону;

- TIME_ZONE: дефолтна часова зона проекту; у нас в проекті вона в UTC; але, якщо у вас в проекті ви точно знаєте, для користувачів якої країни працюватиме ваша аплікація - тоді встановлюйте її в часову зону тієї країни.

Наприкінці даної глави ми також займемось налаштуванням змінної MIDDLEWARE_CLASSES, коли активуватимемо вибір мови базуючись на налаштуваннях браузера відвідувача. Але про це пізніше.

Структура папок та файлів

На завершення налаштувань підготуємо структуру папок та файлів. Єдина дія, яку нам потрібно виконати вручну - це створити папку locale в корені аплікації students:

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students
2 $ mkdir locale
```

Наступні підпапки та файли нам створить команда makemessages:

Генеруємо структуру папок та файлів перекладів

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students
2 $ python ./manage.py makemessages --locale=uk --locale=en
```

Зауважте, що у даному прикладі ми запускаємо скрипт manage.py з папки на рівень вище. Команда makemessages аналізує усі файли рекурсивно починаючи з поточної папки, де її запускають. Тому її потрібно запускати будучи в тій папці, де знаходиться папка locale. Відповідно, ми зайдли в корінь аплікації. Таким чином, manage.py скрипт знаходиться тепер на рівень вище і до нього доступаємося через “./manage.py” шлях.

Після даної команди матимемо усі необхідні вкладені папки в locale разом із, поки порожніми, файлами django.po для кожної із мов: англійської та української.

Ми підготували папку з перекладами лише в аплікації, оскільки сам проект поки не містить жодних стрічок, що відображаються на користувачькому інтерфейсі. Відповідно, перекладати нічого.

На завершення налаштувань потрібно додати файли із розширенням “.mo” в список файлів для ігнорування репозиторіем. Дані файли є генерованими (компільованими) версіями текстових файлів “.po”) і тому їх не варто тримати всередині репозиторію. Для цього поредагуйте файл `.gitignore` в корені репозиторію і переконайтесь, що серед списку шаблонів є шаблон “.mo”.

Тепер можете сміливо закомітити та запушити останні зміни в репозиторій коду. Ми готові переходити до шаблонів.

Перекладаємо шаблони

У нас все готово, щоб переходити до інтернаціоналізації проекту. Почнемо із шаблонів. А саме із шаблону `base.html`. Переклавши його ми зробимо майже 50% усієї справи, адже стрічки даного шаблону відображаються на усіх сторінках проекту.

Перекладаємо шаблон `base.html`

Усі шаблони, що використовують переклади повинні завантажувати модуль `i18n`.

Відкриваємо шаблон `base.html` у редакторі і замінюємо усі стрічки української мови на тег `trans`. Даний тег міститиме переклад української стрічки на англійську мову. Тобто канонічний варіант.

Наведені лише частини шаблону `base.html`, які були змінені. Для повної картини використовуйте код, що йде із книгою:

Інтернаціоналізований base.html шаблон

```
1  {% load i18n %}  
2  {% load static from staticfiles %}  
3  <!DOCTYPE html>  
4  ...  
5      <title>{% trans "Students Accounting Service" %} - {% block meta_t\title %}{% endblock meta_title %}</title>  
6      <meta name="description" value="{% trans "App to manage students d\atabase" %}" />  
7      ...  
8      <!-- Start Header -->  
9      <div class="row" id="header">  
10         <div class="col-xs-8">  
11             <h1>{% trans "Students Accounting Service" %}</h1>  
12         </div>  
13         <div class="col-xs-4" id="group-selector">  
14             <strong>{% trans "Group:" %}</strong>  
15             <select>  
16                 <option value="">{% trans "All Students" %}</option>  
17             ...  
18             <!-- Start subheader -->  
19             <div class="row" id="sub-header">  
20                 <div class="col-xs-12">  
21                     <ul class="nav nav-tabs" role="tablist">  
22                         <li role="presentation" {% if request.path == '/' %}class=\\"active\"{% endif %}><a href="{% url "home" %}">{% trans "Students" %}\</a></li>  
23                         <li role="presentation" {% if '/journal' in request.path %}\<li role="presentation" {% if '/journal' in request.path %}class=\\"active\"{% endif %}><a href="/journal">{% trans "Journal" %}</a></li>  
24                         <li role="presentation" {% if '/groups' in request.path %}\<li role="presentation" {% if '/groups' in request.path %}class=\\"active\"{% endif %}><a href="{% url "groups" %}">{% trans "Gro\ups" %}</a></li>  
25                         <li role="presentation" {% if '/contact-admin' in request.\path %}\<li role="presentation" {% if '/contact-admin' in request.\path %}class=\\"active\"{% endif %}><a href="{% url "contact_admin" %}"\>
```

```
35 >{% trans "Contact" %}</a></li>
36     </ul>
37   </div>
38 </div>
39 <!-- End subheader -->
40 ...
41 <!-- Start Footer -->
42 <div class="row" id="footer">
43   <div class="col-xs-12">
44     &copy; 2014 {% trans "Students Accounting Service" %}</div>
45   </div>
46 </div>
47 <!-- End Footer -->
48 ...
```

Давайте детальніше глянемо на кілька важливих моментів змін:

- 1ий рядок: обов'язково потрібно завантажити модуль i18n; він дасть нам доступ до тегу trans;
- 5ий: частину вмісту тегу title огорнули в тег trans; заголовок нашої аплікації в мета заголовку сторінки тепер буде англійською; пізніше ми його локалізуємо під український варіант; тег trans отримав аргументом статичну стрічку англійською мовою; він також може отримувати аргументом назvu змінної доступну в шаблоні;
- 7ий: подібним чином інтернаціоналізували мета опис сторінки.

Усі наступні куски коду аналогічним чином містять використання тегу trans з переданими англійськими стрічками.

Тепер можете перевантажити домашню сторінку аплікації у веб-переглядачі і переконатись, що заголовок та футер сторінки відображаються англійською мовою.

Якщо після оновлення сторінки ви отримали помилку: “Invalid block tag: ‘trans’, expected ‘endblock’”, тоді, швидше за все, ви забули скористатись тегом load, щоб завантажити модуль i18n.

Ми з вами щойно зробили лише половину справи інтернаціоналізувавши шаблон base.html. Залишилось його локалізувати для україномовних користувачів.

Користуючись командою makemessages згенеруємо файли перекладів, що міститимуть нові стрічки із шаблону base.html:

Оновлюємо файли перекладів

```
1 # важливо зайти в корінь аплікації
2 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students
3
4 # запускаємо команду makemessages, яка оновить файли
5 # перекладів новими стрічками
6
7 # дана команда оновить файли для обидвох мов
8 $ python ./manage.py makemessages --locale=en --locale=uk
```

Ще раз хочу нагадати, що ми запускаємо скрипт manage.py з папки на рівень вище від поточної. Команду makemessages варто запускати в папці, яка містить locale. Данна команда рекурсивно аналізує файли на стрічки перекладу.

Після даної команди отримаємо оновлені файли django.po в обидвох папках: en і uk. Нас цікавить лише підпапка uk, адже там міститься django.po файл, в якому потрібно прописати переклади англійських стрічок на українську. Відкриваємо файл з українською мовою, який знаходиться в підпапці “locale/uk/LC_MESSAGES”:

django.po в папці з українською мовою

```
1 ...
2 #: templates/students/base.html:7
3 #: templates/students/base.html:33
4 msgid "Students Accounting Service"
5 msgstr ""
6
7 #: templates/students/base.html:36
8 msgid "Group:"
9 msgstr ""
10
11 #: templates/students/base.html:38
12 msgid "All Students"
13 msgstr ""
14
15 #: templates/students/base.html:51
16 msgid "Students"
17 msgstr ""
18 ...
```

У вищеприведеному файлі з перекладом ми оминули початок django.po файлу і відобразили лише кілька стрічок для перекладу. В кожній групі параметр msgstr є порожнім. Завданням перекладача є заповнити дані параметри перекладами на українську мову.

Тепер “одягаємо шапку” перекладача і заповнюємо файл перекладами на українську мову:

Файл із заповненими перекладами

```
1 #: templates/students/base.html:7
2 #: templates/students/base.html:33
3 msgid "Students Accounting Service"
4 msgstr "Сервіс Обліку Студентів"
5
6 #: templates/students/base.html:36
7 msgid "Group:"
8 msgstr "Група:"
9
10 #: templates/students/base.html:38
11 msgid "All Students"
12 msgstr "Усі Студенти"
13
14 #: templates/students/base.html:51
15 msgid "Students"
16 msgstr "Студенти"
17
18 #: templates/students/base.html:52
19 msgid "Journal"
20 msgstr "Відвідування"
21
22 #: templates/students/base.html:53
23 msgid "Groups"
24 msgstr "Групи"
25
26 #: templates/students/base.html:54
27 msgid "Contact"
28 msgstr "Контакт"
```

Нагадаємо формат кожного із записів:

- кожна стрічка перекладу складається із кількох рядків; дані групи розділені між собою порожнім рядком;

- перший рядок починається із “#” і вказує на шляхи до файлів, в яких дана стрічка використовується;
- другий рядок - це параметр msgid і позначає ID стрічки або її значення у канонічній мові (основній); в нашому випадку це англійська мова;
- третій рядок - це параметр msgstr і містить стрічку перекладу; в нашому випадку це переклад на українську мову.

Команда, яка збирає стрічки із файлів виконує додаткову функцію: пробує вгадати подібні стрічки і підставляє їх під один переклад. Наприклад: “Усі Студенти” і “Видалити Студента” можуть потрапити під одну стрічку. При цьому ви побачите рядок із ключовим словом fuzzy, який означатиме, що процедура збору файлу перекладу спробувала зібрати подібні слова в одну стрічку перекладу і тепер пропонує вам або погодитись на даний переклад, або, все ж таки, розділити на дві різні стрічки:

Рядок fuzzy у файлах перекладу

```
1 #: templates/students/base.html:36
2 #, fuzzy
3 #:| msgid "Group:"
4 msgid "New Group:"
5 msgstr ""
6
7 #: templates/students/base.html:37
8 msgid "Group:"
9 msgstr "Група:"
```

Вищеприведений приклад стрічки означає, що утиліта gettext знайшла дві, як вона вважає, подібні стрічки “Group:” і “New Group:”. Тому вони були об’єднані в одну стрічку перекладу. Щоб повідомити перекладача про дану дію, у файл перекладу додано рядок “#, fuzzy”. Перекладач повинен вирішити чи розділити дану стрічку на дві, а чи поправити в шаблоні і об’єднати “New Group:” і “Group:” в одну стрічку. Як бачите, рядок ‘#:| msgid “Group:”’ позначає подібні стрічки. Їх може бути декілька.

Ось як може виглядати результат розділу даної стрічки:

Робота з fuzzy стрічками

```
1 #: templates/students/base.html:36
2 msgid "New Group:"
3 msgstr "Нова Група:"
4
5 #: templates/students/base.html:37
6 msgid "Group:"
7 msgstr "Група:"
```

Тобто ви, як перекладач, видаляєте рядки з fuzzy та схожою стрічкою і залишаєте дві різних стрічки для перекладу. В подальшому команда `makemessages` не виконуватиме fuzzy перевірки на дані стрічки.

Після того як пройшлися по усіх стрічках потрібно скомпілювати наші файли перекладу:

Компілюємо файли перекладу

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students
2 $ python ../manage.py compilemessages
```

В результаті отримаєте оновлені файлики “*.mo” в підпапках `locale`. Рестартуйте Django сервер, оновлюйте домашню сторінку у браузері і тестуйте результат. На даний момент шапка та футер сторінки повинні бути перекладеними на українську мову.

Перекладаємо домашню сторінку

Щоб завершити переклад домашньої сторінки, нам потрібно зайнятись шаблоном `students_list.html`. Тому відкриваємо його в редакторі і шукаємо усі стрічки українською мовою:

Блоки шаблону students_list.html з українською мовою

```
1 ...
2 {% block meta_title %}Студенти{% endblock meta_title %}
3 ...
4 {% block title %}База Студентів{% endblock title %}
5 ...
6 <a href="{% url "students_add" %}" class="btn btn-primary">Додати Ст\удента</a>
7 ...
8 ...
9     <th>Фото</th>
10 ...
11     <th>
12         <a href="{% url "home" %}?order_by=last_name{% if order_by \&lt;= 'last_name' and reverse != '1' %}&amp;reverse=1{% endif %}">
13             Прізвище
14             {% if order_by == 'last_name' and reverse != '1' %}&uarr;;
15 ...
16     <th>
17         <a href="{% url "home" %}?order_by=first_name{% if order_by \&lt;= 'first_name' and reverse != '1' %}&amp;reverse=1{% endif %}">
18             Ім'я
19             {% if order_by == 'first_name' and reverse != '1' %}&uarr;;
20             {% elif order_by == 'first_name' and reverse == '1' %}&darr;;
21             rr;
22             {% endif %}
23 ...
24     </th>
25 ...
26     <th>
27         <a href="{% url "home" %}?order_by=ticket{% if order_by == 'ticket' and reverse != '1' %}&amp;reverse=1{% endif %}">
28             № Билету
29             {% if order_by == 'ticket' and reverse != '1' %}&uarr;;
30             {% elif order_by == 'ticket' and reverse == '1' %}&darr;;
31             {% endif %}
32 ...
33     </th>
```

```
35      <th>Дії</th>
36 ...
37      <div class="btn-group">
38          <button type="button" class="btn btn-default dropdown-toggle"
39          gle="
40              data-toggle="dropdown">Дія
41              <span class="caret"></span>
42          </button>
43 ...
44          <ul class="dropdown-menu" role="menu">
45              <li><a href="{% url "students_edit" student.id %}" clas\
46 s="student-edit-form-link">Редагувати</a></li>
47              <li><a href="{% url "journal" student.id %}">Відвідуван\
48 ня</a></li>
49              <li><a href="{% url "students_delete" student.id %}">Ви\
50 далити</a></li>
51          </ul>
52 ...
```

Кожен із наведених блоків коду містить стрічку з українською мовою і потребує заміни на тег trans. Подібним чином до того, як ми це робили в шаблоні base.html, тут також оновлюємо дані блоки.

Ось як виглядатимуть оновлені частини коду:

students_list.html інтернаціоналізований

```
1  {% extends "students/base.html" %}
2
3  {% load i18n %}
4  {% load static from staticfiles %}
5 ...
6  {% block meta_title %}{% trans "Students" %}{% endblock meta_title %}
7 ...
8  {% block title %}{% trans "Students List" %}{% endblock title %}
9 ...
```

```
10 <a href="#"><% url "students_add" %>" class="btn btn-primary">% trans \
11 "Add Student" %}</a>
12 ...
13     <th>% trans "Photo" %</th>
14 ...
15     <th>
16         <a href="#"><% url "home" %>?order_by=last_name{% if order_by \
17 == 'last_name' and reverse != '1' %}&reverse=1{% endif %}">
18             {% trans "Last Name" %}
19             {% if order_by == 'last_name' and reverse != '1' %}&uarr;;
20 ...
21     <th>
22         <a href="#"><% url "home" %>?order_by=first_name{% if order_by \
23 == 'first_name' and reverse != '1' %}&reverse=1{% endif %}">
24             {% trans "First Name" %}
25             {% if order_by == 'first_name' and reverse != '1' %}&uarr;;
26             {% elif order_by == 'first_name' and reverse == '1' %}&darr\
27 rr;
28             {% endif %}
29     </th>
30 ...
31     <th>
32         <a href="#"><% url "home" %>?order_by=ticket{% if order_by == \
33 'ticket' and reverse != '1' %}&reverse=1{% endif %}">
34             {% trans "Ticket #" %}
35             {% if order_by == 'ticket' and reverse != '1' %}&uarr;;
36             {% elif order_by == 'ticket' and reverse == '1' %}&darr;;
37             {% endif %}
38     </th>
39     <th>% trans "Actions" %</th>
40 ...
41     <div class="btn-group">
42         <button type="button" class="btn btn-default dropdown-tog\
43 gle"
44             data-toggle="dropdown">% trans "Action" %}
```

```
45          <span class="caret"></span>
46      </button>
47  ...
48      <ul class="dropdown-menu" role="menu">
49          <li><a href="{% url "students_edit" student.id %}" clas\
50 s="student-edit-form-link">{% trans "Edit" %}</a></li>
51          <li><a href="{% url "journal" student.id %}">{% trans "\\
52 Journal" %}</a></li>
53          <li><a href="{% url "students_delete" student.id %}">{%\\
54 trans "Delete" %}</a></li>
55      </ul>
56  ...
```

Зверніть увагу, що на третьому рядку ми також додали тег load, який завантажує модуль i18n.

Тег extends завжди повинен бути першим тегом в шаблоні. Якщо його переставити місцями із тегом load, тоді отримаємо помилку.

Інтернаціоналізувавши шаблон students_list.html, знову запускаємо команду для оновлення файлів перекладів. Команда не перетре наших ручних змін, а лише оновить новими стрічками та шляхами на оновлені місця у файлах:

Оновлюємо файли перекладів

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students
2 $ python ../manage.py makemessages --locale=en --locale=uk
```

Відкриваємо файл перекладу для української мови django.po та заповнюємо новододані порожні стрічки:

Оновлення файлу locale/uk/LC_MESSAGES/django.po

```
1 ...
2 #: templates/students/base.html:51 templates/students/students_list.\
3 html:6
4 msgid "Students"
5 msgstr "Студенти"
6
7 #: templates/students/base.html:52 templates/students/students_list.\
8 html:76
9 msgid "Journal"
10 msgstr "Відвідування"
11
12 #: templates/students/students_list.html:8
13 msgid "Students List"
14 msgstr "Список Студентів"
15
16 #: templates/students/students_list.html:11
17 msgid "Add Student"
18 msgstr "Додати Студента"
19
20 #: templates/students/students_list.html:19
21 msgid "Photo"
22 msgstr "Фото"
23
24 #: templates/students/students_list.html:22
25 msgid "Last Name"
26 msgstr "Прізвище"
27
28 #: templates/students/students_list.html:30
29 msgid "First Name"
30 msgstr "Ім'я"
31
32 #: templates/students/students_list.html:37
33 msgid "Ticket #"
34 msgstr "№ Білету"
```

```
35
36 #: templates/students/students_list.html:42
37 msgid "Actions"
38 msgstr "Дії"
39
40 #: templates/students/students_list.html:71
41 msgid "Action"
42 msgstr "Дія"
43
44 #: templates/students/students_list.html:75
45 msgid "Edit"
46 msgstr "Редагувати"
47
48 #: templates/students/students_list.html:77
49 msgid "Delete"
50 msgstr "Видалити"
51 ...
```

Команда makemessages зібрала для нас нові стрічки, а також оновила існуючі, які були спільними для обидвох шаблонів: base.html та students_list.html. Так, наприклад, у першому рядку вищепереліченого прикладу бачимо, що стрічка Students міститься і в шаблоні base.html, і в шаблоні students_list.html.

Вищеперелічений приклад містить стрічки уже із заповненими параметрами msgstr, а також із відредактованими вручну fuzzy записами.

Перед тестом нових перекладів потрібно їх скомпілювати:

Компілюємо файли перекладу

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students
2 $ python ../manage.py compilemessages
```

Перевантажуємо сервер Django, оновлюємо домашню сторінку аплікації у веб-переглядачі і тестуємо переклади.

Перекладаємо сторінку видалення групи

На завершення роботи із шаблонами, розглянемо сторінку видалення групи. Це дозволить трохи попрактикуватись із тегом blocktrans.

Ось як на даний момент виглядає наш шаблон підтвердження видалення групи:

Шаблон groups_confirm_delete.html

```
1  {% extends "students/base.html" %}  
2  
3  {% load static from staticfiles %}  
4  
5  {% block meta_title %}Видалити Групу{% endblock meta_title %}  
6  
7  {% block title %}Видалити Групу{% endblock title %}  
8  
9  {% block content %}  
10  
11 <form action="{% url "groups_delete" object.id %}" method="post">  
12   {% csrf_token %}  
13   <p>Ви дійсно хочете видалити групу "{{ object }}"?</p>  
14   <input type="submit" value="Так" name="delete_button"  
15     class="btn btn-danger" />  
16   <a href="{% url "groups" %}" class="btn btn-info">Ні</a>  
17   <br /><br />  
18 </form>  
19  
20  {% endblock content %}
```

Для його інтернаціоналізації ми, як завжди, маємо додати завантаження модуля i18n, а також замінити усі кириличні стрічки англомовними аналогами та огорнути у тег trans:

Інтернаціоналізований шаблон groups_confirm_delete.html

```
1  {% extends "students/base.html" %}  
2  
3  {% load i18n %}  
4  {% load static from staticfiles %}  
5  
6  {% block meta_title %}{% trans "Delete Group" %}{% endblock meta_tit\  
7  le %}  
8  
9  {% block title %}{% trans "Delete Group" %}{% endblock title %}  
10  
11  {% block content %}  
12  
13  <form action="{% url "groups_delete" object.id %}" method="post">  
14      {% csrf_token %}  
15      <p>  
16          {% blocktrans %}Do you really want to delete group: "{{ object }}\  
17  }" ?{% endblocktrans %}  
18      </p>  
19      <input type="submit" value="{% trans "Yes" %}" name="delete_button"  
20          class="btn btn-danger" />  
21      <a href="{% url "groups" %}" class="btn btn-info">{% trans "No" %}\<br>  
22  </a>  
23      <br /><br />  
24  </form>  
25  
26  {% endblock content %}
```

Є кілька новинок у вищезгаданій інтернаціоналізації шаблону:

- 16ий рядок: для перекладу повідомлення про видалення групи ми скористались тегом blocktrans; він тут обов'язковий, адже текст повідомлення містить динамічну змінну object; тег trans не дозволяє поєднання змінних і статичних стрічок;

- 19ий: тег `trans` можемо використовувати будь-де всередині HTML тегів; в даному випадку він використовується, щоб вставляти перекладені стрічки в якості атрибутів HTML тегу `input`.

Важливо пам'ятати, що `blocktrans` не може містити тегу `url`, а також інші визначення змінних. Всередині даного тегу ми можемо лише вставляти змінні, що уже є визначеними або до тегу `blocktrans`, або в означені даного тегу з допомогою ключового слова `with`. Більше про це пояснено в теоретичній частині даної глави.

Наступні кроки по локалізації даного шаблону є аналогічними:

- `makemessages`, щоб оновити файли перекладу;
- вручну перекладаємо нові стрічки;
- компілюємо файли перекладу з допомогою команди `compilemessages`;
- рестарт сервера та перевірка сторінки.

Також `blocktrans` тег підтримує переклад множини та однини, але практичної потреби у даній властивості у нас немає в межах цього проекту.

...

На цьому закінчуємо роботу із адаптацією шаблонів і переходимо до інтернаціоналізації Python коду.

На домашнє завдання попрошу вас пройтись по усіх шаблонах проекту і здійснити подібний переклад. Не весь графічний інтерфейс вдастся перекласти з першої спроби, адже частина стрічок “зашиті” в Python коді. Про це поговоримо в наступній секції.

В коді, що йде з книгою можна звіритись із результатами власного перекладу. Проект бази студентів повністю перекладений на дві мови: англійську та українську.

Перекладаємо Python код

Частина стрічок, що виводяться на інтерфейс користувача генеруються Python кодом. Тому, щоб повністю перекласти веб-інтерфейс аплікації, маємо також зайнятись перевіркою в кожному Python модулі, що може містити кириличні стрічки.

Python модулів, як і загалом коду, в аплікації у нас уже трохи назбиралось. Тому ми пройдемось лише по тих місцях, які продемонструють основні моменти інтернаціоналізації Python коду на практиці, а решту коду залишиться на домашнє завдання.

Як визначати чи модуль містить не ASCII³⁷³ символи? Дуже просто. Забираєте заголовок utf-8 із Python модуля, оновлюєте сторінку у веб-браузері, за яку відповідає даний модуль і, якщо з'явилася помилка - значить у модулі є над чим попрацювати.

Із теоретичної частини даної глави ми уже знаємо, що процедура адаптації та перекладу стрічок в Python коді є дуже подібною до перекладу шаблонів:

- переглядаємо код і замінюємо кирилицю на англійську мову;
- також огортаємо англійські стрічки у функції перекладу (в шаблонах ми мали справу із тегами перекладу);
- оновлюємо файли перекладу з допомогою команди `makemessages`;
- редактуємо файли перекладу і перекладаємо нові англійські стрічки на українську мову; при цьому беремо до уваги стрічки `fuzzy`, які для нас спробувала вгадати утиліта `gettext`;

³⁷³<https://ru.wikipedia.org/wiki/ASCII>

- компілюємо файли перекладу з допомогою команди compilemessages;
- рестартуємо Django сервер, оновлюємо сторінку у браузері, тестируємо результат;
- якщо все добре, тоді закидуємо зміни в репозиторій.

Тому в подальших секціях я не наводитиму команд makemessages чи compilemessages, а лише згадуватиму, коли їх варто буде робити. Це зекономить нам кілька сторінок книги, які ми використаємо ефективніше.

Починаємо нашу роботу із в'юшкі, що віддає домашнюю сторінку:

В'юшка домашньої сторінки

Відкриваємо в редакторі Python модуль, де знаходяться в'юшки домашньої сторінки і управління студентами: “views/students.py”. Видаляємо заголовок “`# -- coding: utf-8 --`” і перезавантажуємо головну сторінку.

Результатом буде подібна помилка в консолі, де ви запустили команду runserver:

Помилка через брак заголовка з кодуванням файлу

```
1 SyntaxError: Non-ASCII character '\xd1' in file /data/work/virtualenv\
2 vs/studentsdb/src/studentsdb/students/views/students.py on line 18, \
3 but no encoding declared; see http://python.org/dev/peps/pep-0263/ f\
4 or details
```

Тепер ми точно знаємо, що наш модуль містить не ASCII символи, яких нам потрібно позбутись. Давайте виділимо кілька блоків даного модуля із кириличними стрічками:

Приклади стрічок коду із не ASCII символами

```
1 ...
2         if not first_name:
3             errors['first_name'] = u"Ім'я є обов'язковим"
4         else:
5             data['first_name'] = first_name
6 ...
7         if not ticket:
8             errors['ticket'] = u"Номер білета є обов'язковим"
9         else:
10            data['ticket'] = ticket
11 ...
12         # redirect to students list
13         return HttpResponseRedirect(
14             u'%s?status_message=Студента успішно додано!' %
15             reverse('home'))
16 ...
17     # add buttons
18     self.helper.layout[-1] = FormActions(
19         Submit('add_button', u'Зберегти', css_class="btn btn-primary"),
20         Submit('cancel_button', u'Скасувати', css_class="btn btn-link"),
21         )
22 ...
23     )
24 ...
25     def get_success_url(self):
26         return u'%s?status_message=Студента успішно збережено!' %
27             reverse('home')
28 ...
```

Вище ми навели куски коду, які взяті із різних класів та функцій модуля `students.py`. Кожен із них містить стрічку українською мовою. Даний приклад містить не усі стрічки для перекладу, а лише ту частину, яка відображає основні маніпуляції із стрічками в Python.

Другим кроком інтернаціоналізації Python модуля після видалення заголовка utf8 кодування буде імпорт функції перекладу:

Імпортуємо функцію перекладу

```
1 from django.utils.translation import ugettext as _
```

Даний імпорт додаємо на початку модуля серед інших імпортів із кореневого пакету django.

Ми готові до використання функції “`_`” на практиці. Ось вищезгадані блоки коду, але після застосування інтернаціоналізації:

i18n в Python стрічках

```
1 ...
2 from django.forms import ModelForm
3 from django.views.generic import UpdateView, DeleteView
4 from django.utils.translation import ugettext as _
5 ...
6 ...
7         if not first_name:
8             errors['first_name'] = _("First Name field is requ\ \
9 ired")
10        else:
11            data['first_name'] = first_name
12 ...
13        if not ticket:
14            errors['ticket'] = _("Ticket number is required")
15        else:
16            data['ticket'] = ticket
17 ...
18        # redirect to students list
19        return HttpResponseRedirect(
20            u'%s?status_message=%s' % (reverse('home'),
21            _(u"Student added successfully!")))
22 ...
```

```
23         # add buttons
24         self.helper.layout[-1] = FormActions(
25             Submit('add_button', _(u'Save'), css_class="btn btn-pri\
26 mary"),
27             Submit('cancel_button', _(u'Cancel'), css_class="btn bt\
28 n-link"),
29         )
30 ...
31     def get_success_url(self):
32         return u'%s?status_message=%s' % (reverse('home'),
33                                         _(u"Student updated successfully!"))
34 ...
```

Давайте детальніше пройдемось по кількох важливих рядках коду:

- 4ий рядок: імпортуємо функцію gettext для подальшого використання;
- 8ий: перекладаємо повідомлення про помилку в полі Ім'я на англійську і передаємо цю стрічку аргументом функції gettext (під назвою “_”); таким чином, під ключем ‘first_name’ в словнику errors ми отримаємо уже перекладену стрічку про помилку відповідно до активної мови;
- 20ий: в статусне повідомлення передаємо також уже перекладену стрічку повідомлення про успішно створеного студента; обов'язково усі аргументи даної операції повинні бути юнікодовими стрічками (або містити лише ascii символи); при активній українській мові кінцева стрічка міститиме кирилицю, а тому обов'язково має бути представлена в кодіapplікації як юнікодова стрічка; в протилежному випадку отримаємо помилку;
- 25ий: тут ми інтернаціоналізуємо кнопки Зберегти та Скасувати на формі додавання студента;
- 32ий: переклад статусного повідомлення про успішно оновленого студента; тут головне не пропустити кілька закриваючих дужок наприкінці стрічки з перекладом.

Повністю перегляньте код даного модуля і адаптуйте усі стрічки допоки головна сторінка веб-applікації не перестане ламатись у веб-переглядачі. Лише

після цього переходить до оновлення файлів перекладу з допомогою команди makemessages.

Після запуску makemessages аналогічним чином як ми це робили попередні рази, відкриваємо файл перекладу на українську мову і заповнюємо порожні стрічки:

Частина оновлених стрічок у файлі django.po

```
1 ...
2 #: models/groups.py:9 models/students.py:44 templates/students/base.\
3 html:32
4 #: templates/students/students_add.html:79
5 msgid "Group"
6 msgstr "Група"
7 ...
8 #: models/groups.py:25 models/students.py:51
9 #: templates/students/students_add.html:92
10 msgid "Extra Notes"
11 msgstr "Додаткові Нотатки"
12 ...
13 #: models/students.py:10 templates/students/base.html:47
14 #: templates/students/students_list.html:5
15 msgid "Students"
16 msgstr "Студенти"
17 ...
18 #: templates/students/students_add.html:104 views/groups.py:71
19 #: views/students.py:148
20 msgid "Cancel"
21 msgstr "Скасувати"
22 ...
23 #: views/students.py:56
24 msgid "First Name field is required"
25 msgstr "Ім'я є обов'язковим"
26 ...
27 #: views/students.py:108
28 msgid "Student added successfully!"
```

```
29 msgstr "Студента додано успішно!"  
30 ...
```

Вище наведена лише частина нових і оновлених стрічок базуючись на змінах в `students.py` модулі. Дані стрічки, як можете зауважити, уже є перекладеними. Це ваше завдання їх заповнити. В реальних проектах цим, зазвичай, займатиметься окрема людина - перекладач.

Коли закінчили із перекладом, знову компілюєте файли перекладу і рестартуєте Django сервер. Тепер форма додавання студента повинна містити частину стрічок із файлу перекладу. Спробуйте запостити форму з помилками і без них, а також натиснути кнопку Скасувати на формі. Переконайтесь, що статусні повідомлення щодо ваших дій є перекладені на українську мову.

Якщо ви вже виконали попереднє домашнє завдання щодо перекладу решти шаблонів у проекті, тоді ваша форма додавання містить усі кириличні стрічки, що знаходяться у файлі перекладу. Таким чином, якщо ви й надалі використовуєте форму додавання, що є повністю вручну написана на статичному HTML коді - форма повністю інтернаціоналізована. Проте, якщо в процесі роботи над домашніми завданнями, ви мігрували дану форму на форму моделей (аналогічно до форми редагування студента), тоді поля даної форми й надалі міститимуть українські стрічки, що приходять напряму із полів класу моделі. Цим ми займемось пізніше.

В наступній секції ми розберемось із використанням варіанту “`_lazy`” функції `gettext`. А саме, інтернаціоналізуємо моделі.

Перекладаємо моделі

Ми розібралися як перекладати стрічки в коді в’юшок. В даній секції розглянемо переклад стрічок, що лежать на рівні модуля або є атрибутами класу. Тобто такий код, який запускається лише один раз - під час запуску Django сервера. В таких випадках потрібно працювати із “`_lazy`” функціями.

Почнемо із моделі студента. Відкриваємо модуль ‘models/students.py’ і першим ділом видаляємо заголовок із кодуванням файлу “# -- coding: utf-8 --”. Наступним кроком додаємо імпорт функції ugettext_lazy:

Модель студента

```
1 ...
2 from django.db import models
3 from django.utils.translation import ugettext_lazy as _
4
5
6 class Student(models.Model):
7     """Student Model"""
8
9     class Meta(object):
10         verbose_name = u"Студент"
11         verbose_name_plural = u"Студенти"
12
13     first_name = models.CharField(
14         max_length=256,
15         blank=False,
16         verbose_name=u"Ім'я")
17
18     last_name = models.CharField(
19         max_length=256,
20         blank=False,
21         verbose_name=u"Прізвище")
22 ...
```

З цього невеликого фрагменту коду можемо бачити, що у нас є кириличні стрічки на рівні атрибутів вкладеного класу Meta. Також кожне із полів моделі Student включає атрибут verbose_name, який потребує інтернаціоналізації.

Функцію ugettext_lazy ми також імпортували під іменем “_”. Час нею скористатись:

Модель студента

```
1 ...
2 from django.db import models
3 from django.utils.translation import ugettext_lazy as _
4
5
6 class Student(models.Model):
7     """Student Model"""
8
9     class Meta(object):
10         verbose_name = _(u"Student")
11         verbose_name_plural = _(u"Students")
12
13     first_name = models.CharField(
14         max_length=256,
15         blank=False,
16         verbose_name=u"Ім'я")
17
18     last_name = models.CharField(
19         max_length=256,
20         blank=False,
21         verbose_name=_(u"Last Name"))
22 ...
```

Ззовні робота із функцією `ugettext_lazy` нічим не відрізняється від того, як працює функція `ugettext`. Основна відмінність спостерігатиметься, якщо після старту Django сервера спробувати переключатись між українською та англійською мовами на сайті. Мови не переключатимуться на рівні полів моделі студента, якщо не скористаємося варіантом “`_lazy`”, адже клас моделі студента і його атрибути класу (у нашому випадку це поля моделі) вичитуються інтерпретатором лише на старті сервера при першому імпорті модуля. Функція `ugettext` одразу перекладе передану стрічку згідно дефолтної мови і запам'ятаває в атрибуті поля.

Натомість функція `ugettext_lazy` передасть спеціальний об'єкт в атрибут класу моделі, який здійснюватиме переклад канонічної стрічки лише в момент

його використання в шаблоні. Таким чином завжди працюючи від поточної активованої мови для користувача.

Ми інтерніціоналізували лише кілька перших полів моделі студента. Перед тим як рухатись далі, пройдіться по решті полів і переконайтесь, що вони готові до перекладів. Інакше модуль моделі студента просто не запуститься, адже ми забрали заголовок із utf-8 кодуванням. Для перевірки правильності власних дій можете звіритись із кодом, що йде разом із книгою.

Далі, уже по традиції, перезбираємо файли перекладів з допомогою команди makemessages. Відкриваємо файл з українською мовою django.po і заповнюємо новододані порожні стрічки:

Перекладаємо поля моделі студента

```
1 ...
2 #: models/students.py:51
3 msgid "Extra Notes"
4 msgstr "Додаткові Нотатки"
5 ...
6 #: models/students.py:10 templates/students/base.html:47
7 #: templates/students/students_list.html:5
8 msgid "Students"
9 msgstr "Студенти"
10 ...
11 #: models/students.py:15
12 #: templates/students/students_list.html:29
13 msgid "First Name"
14 msgstr "Ім'я"
15 ...
16 #: models/students.py:20
17 #: templates/students/students_list.html:21
18 msgid "Last Name"
19 msgstr "Прізвище"
20 ...
```

Велика частина стрічок співпала із тими, що ми уже переклали в попередніх секціях при роботі із шаблонами. Тому нам прийдеться перекласти значно менше нових стрічок, аніж ми інтернаціоналізували в модулі із класом моделі студента.

Коли закінчите із перекладом усіх порожніх стрічок, правкою fuzzy повідомлень, тоді переходіть до компіляції файлів перекладів з допомогою команди `compilemessages`.

Рестарт сервера, оновлення сторінки в браузері, і ми готові до тесту форми редактування студента. Тепер вона повинна містити усі назви полів форми, що беруться із файлу `django.po`.

Сервіс Обліку Студентів

Редагувати Студента

Список Студентів

Додати Студента

#	Фото	Прізвище	Помітка
1		Лісовий	Підготуваний
2		Лісянка	Підготуваний
3		Кеня	Підготуваний

« 1 2 3 4 5 6 »

© 2015 Сервіс Обліку Студентів

Ім'я*

Прізвище*

По-батькові

Дата народження*

Фото No file chosen

Білет*

Група*

Дії

Дія

Дія

Дія

Зберегти **Скасувати**

Форма редактування студента після локалізації моделі

На формі редактування студента ми отримали локалізовані поля. Але крім цього можемо бачити, що кнопки форми Зберегти та Скасувати також перевкладені. Їх ми адаптували ще на попередньому кроці, коли працювали із в'юшками студента. А заголовок форми “Редагувати Студента” поки береться із шаблона і є напряму в нього “вшитий” без задіювання файлів перекладів. Це залишається вам на домашнє завдання.

Дні в журналі відвідування

Наступним кроком перейдемо до сторінки із журналом відвідування. Якщо пам'ятаєте в шаблоні журналу відвідування ми залишили “TODO” нотатки:

Доробки в журналі відвідування

```

1   <thead>
2     <tr>
3       <th>#</th>
4       <th>Студент</th>
5       {% for day in month_header %}
6         <th class="day-header">
7           <!-- TODO: translate day name -->
8           {{ day.verbose }}
9           <br />
10          {{ day.day }}
11        </th>
12      {% endfor %}
13    </tr>
14  </thead>

```

Рядок із TODO вказує, що потрібно перекласти ім'я тижня. Зраз у нас лише англійський варіант скороченої назви дня тижня:

		Зміни в журнал зберігаються автоматично при кожному кліку в клітинці таблиці.																											
		← February 2015 →																											
#	Студент	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa
11	Майнінг 3 Віктор	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□

Дні, які потрібно перекласти

Щоб вирішити дану проблему маємо два варіанти:

- поцікавитись чи Django фреймворк уже містить переклади днів тижня і, якщо так, тоді скористатись ними;
- самостійно перекласти усі дні тижня у наших файлах перекладу.

Більшість популярних фреймворків та систем управління веб-вмістом містять переклади стандартизованих імен: місяців, днів тижня, форматування дати і часу. Не виключенням є і фреймворк Django. Якщо пошукати серед файлів коду фреймворку за файлами із розширенням “*.po”, що містять слово “Monday”, тоді виявите, що більшість файлів django.po в підпапках “site-packages/django/conf/locale” містять не лише переклади повних імен тижнів, але й скорочені варіанти.

На лінуксовых системах шлях до Django коду буде подібним на наступний “/шлях/до/віртуального/середовища/lib/python2.7/site-packages/django”. Це при умові, що ми працюємо із Python версії 2.7. Якщо проблемно знайти дану папку із Django кодом, тоді зайдіть в корінь вашого віртуального середовища і просто здійсніть пошук (використовуючи find команду в командній стрічці на Лінуксі або форму пошуку у дефолтному графічному браузері файлів на вашій операційній системі) за папкою під назвою “site-packages”. Вона містить усі додаткові бібліотеки заінсталювані у ваше віртуальне середовище. Включно із фреймворком Django.

Але дані скорочені варіанти включають у себе три латинські літери дня тижня: Mon, Tue, Wed, Thu. А в нашому журналі відвідування ми відобразили скорочення днів у два символи:

Блок коду із методу `get_context_data` класу `JournalView` в модулі `views/journal.py`

```
1 ...
2         # prepare variable for template to generate
3         # journal table header elements
4         myear, mmonth = month.year, month.month
5         number_of_days = monthrange(myear, mmonth)[1]
6         context['month_header'] = [{'day': d,
7             'verbose': day_abbr[weekday(myear, mmonth, d)][:2]}
8             for d in range(1, number_of_days+1)]
9 ...
```

Як бачите, на передостанньому рядку коду даного блоку ми обрізаемо назву

дня тижня до двох символів: “day_abbr[weekday(myear, mmonth, d)][:2]”. Django фреймворк не надає перекладу для такого варіанту абревіатур. Тому нам потрібно переключитись на використання тризначних скорочень днів тижня і скористатись перекладами, що заклали розробники фреймворку до нас:

Відображаємо трьохзначні дні тижня

```
1 ...
2         # prepare variable for template to generate
3         # journal table header elements
4         myear, mmonth = month.year, month.month
5         number_of_days = monthrange(myear, mmonth)[1]
6         context['month_header'] = [{ 'day': d,
7             'verbose': day_abbr[weekday(myear, mmonth, d)][:3]}
8             for d in range(1, number_of_days+1)]
9 ...
```

Все, що ми зробили - замінили число 2 на 3 при вкороченні стрічки із назвою тижня. Цього достатньо, щоб переходити до шаблону і виконати останній крок в перекладі шапки таблиці журналу.

День тижня в шапці таблиці журналу відвідування відображається напряму в шаблоні, без інтернаціоналізації. Тому відкриваємо шаблон journal.html і додаємо необхідні теги:

Інтернаціоналізуємо день тижня в шаблоні journal.html

```
1 {% extends "students/base.html" %}
2
3 {% load i18n %}
4 {% load static from staticfiles %}
5 ...
6 <thead>
7     <tr>
8         <th>*</th>
9         <th>Студент</th>
10        {% for day in month_header %}
```

```
11      <th class="day-header">
12          {% trans day.verbose %}
13          <br />
14          {{ day.day }}
15      </th>
16      {% endfor %}
17      </tr>
18  </thead>
19 ...
```

В 3-му рядку ми додали тег завантаження модуля i18n. Тому маємо доступ до тегу trans. В рядку 12-му ми оторнули змінну “day.verbose” в тег trans, що застосує знайдені переклади дня тижня згідно активної мови. Також ми забрали нотатку TODO.

Оскільки ми адаптувались під існуючі переклади днів тижня із фреймворку Django, нам не потрібно нічого перекладати власними руками і оновлювати файли перекладу.

Якщо ви спробуєте тепер оновити сторінку Відвідування, отримаєте перекладені дні тижня на українську. А скорочені варіанти імен днів міститимуть 2 та 3 символи згідно Django перекладів: Нед, Пн, Вт, Сер і т.д.

...

В процесі інтернаціоналізації Python коду ми розібрали усі основні випадки використання функцій перекладу. Щоб довести справу до кінця потрібно, використовуючи подібні принципи, пройтись по усіх модулях і провести подібну адаптацію стрічок.

Залишаємо на домашнє опрацювання решту Python коду. Щоб не пропустити жодної кириличної стрічки в Python модулі, просто забираєте utf-8 заголовок і дивитесь чи немає помилки. В будь-якому випадку, можете завжди звіритись із кодом, що йде разом з книгою. Він містить увесь

інтернаціоналізований Python код.

Перекладаємо Javascript код

Останнім місцем, де у нашому проекті можуть міститись кириличні стрічки є Javascript файли. Ми маємо лише один кастомний Javascript файл: main.js. I у ньому маємо три однакових стрічки українською мовою:

Блоки коду із файлу main.js

```
1 ...
2 function initEditStudentPage() {
3 ...
4     'success': function(data, status, xhr){
5         // check if we got successfull response from the server
6         if (status != 'success') {
7             alert('Помилка на сервері. Спробуйте будь-ласка пізніше.');
8             return false;
9         }
10 ...
11     'error': function(){
12         alert('Помилка на сервері. Спробуйте будь-ласка пізніше.');
13         return false
14     }
15 ...
16 function initEditStudentForm(form, modal) {
17 ...
18     form.ajaxForm({
19         'dataType': 'html',
20         'error': function(){
21             alert('Помилка на сервері. Спробуйте будь-ласка пізніше.');
22             return false;
```

```
23     },
24 ...
```

Завдяки теоретичній частині даної глави ми уже знаємо обмеження на стороні клієнта. Javascript код немає доступу ні до потрібних функцій перекладу, ні до файлів з перекладами.

Проте Django фреймворк підготував для нас в'юшку javascript_catalog і вирішив найважчую частину: надав доступ до функцій та файлів перекладу із Javascript коду. Все, що нам потрібно зробити це скористатись даною в'юшкою і зареєструвати новий URL шаблон. Через дану URL адресу ми доступатимемося до необхідного Javascript коду згенерованого в'юшкою javascript_catalog.

Для цього відкриваємо в корені проекту модуль urls.py і додаємо новий URL шаблон:

```
1 ...
2 js_info_dict = {
3     'packages': ('students',),
4 }
5 ...
6 urlpatterns = patterns('',
7 ...
8     url(r'^jsi18n\.js$', 'django.views.i18n.javascript_catalog', js_\
9     info_dict),
10 ...
11 )
12 ...
```

В нашому модулі urls.py ми визначили змінну js_info_dict, яку передамо новому URL шаблону. Даний словник вказує на список пакетів, переклади яких необхідно включити разом із Javascript файлом.

Тепер ми матимемо доступ до функцій та файлів перекладу аплікації students. І це все завдяки Javascript файлу під адресою “/jsi18n.js”.

Щоб надати доступ до змінних даного Javascript файлу нашому кастомному Javascript коду, ми повинні вставити скрипт “jsi18n.js” на кожну сторінку нашої аплікації. Для цього, звісно, використаємо шаблон base.html:

Вставляємо скрипт jsi18n.js у кожну сторінку аплікації, base.html

```
1 ...
2 <script src="http://cdnjs.cloudflare.com/ajax/libs/bootstrap-datetime\ 
3 imepicker/4.0.0/js/bootstrap-datetimepicker.min.js"></script>
4
5 <script src="{% url "django.views.i18n.javascript_catalog" %}"></s\ 
6 cript>
7
8 <script src="{% static "js/main.js" %}"></script>
9 ...
```

Ми помістили новий скрипт між datetimepicker.min.js та нашим main.js файлами.

Варто зауважити одну цікаву річ. В url тезі можемо передавати не лише name аргумент URL шаблона, але й стрічку-шлях до в'юшки, що обслуговує дану адресу. Ту стрічку, яку ми передали в URL шаблоні.

Після останнього кроку на кожній сторінці наш Javascript код матиме доступ до функції gettext. Давайте нею скористаємось. Вона працює подібним чином, як функція ugettext в Python:

Інтернаціоналізуємо Javascript код

```
1 ...
2 function initEditStudentPage() {
3 ...
4     'successfunction(data, status, xhr){
5         // check if we got successfull response from the server
6         if (status != 'success') {
7             alert(gettext('There was an error on the server. Please, t\
8 ry again a bit later.'));
9             return false;
10        }
11 ...
12     'errorfunction(){
13         alert(gettext('There was an error on the server. Please, t\
14 ry again a bit later.'));
15         return false
16     }
17 ...
18 function initEditStudentForm( form, modal) {
19 ...
20     form.ajaxForm({
21         'dataTypeerrorfunction(){
23             alert(gettext('There was an error on the server. Please, try\
24 again a bit later.'));
25             return false;
26         },
27     ...
```

Кожній функції alert, що містила стрічку українською мовою ми тепер переводимо стрічку англійською мовою огорнувши її у виклик функції gettext.

Якщо зараз спробуєте перевантажити вашу сторінку (варто зробити фарс-релоад) і, перед натисканням на лінк форми редагування студента, вимкнете

Django сервер, щоб спричинити спеціальну помилку, то в браузері отримаєте попап вікно із повідомленням англійською мовою.

Ми ще поки не локалізували даних англійських стрічок під українську мову. Для цього треба виконати наші, мабуть уже завчені, дії. Але в даному випадку є одна невелика відмінність: домен для Javascript стрічок називається не django, а djangojs:

Генеруємо файли djangojs.po

```
1 $ python ../manage.py makemessages -d djangojs -l uk -l en
```

Цього разу ми передали команді makemessages додаткову опцію “d”. Вона вказує на домен Javascript перекладів djangojs. Даної команді збере стрічки із наших Javascript файлів і помістить у файли djangojs.po всередині мовних папок.

Ось уже перекладена наша єдина стрічка з англійської на українську мову:

uk/LC_MESSAGES/djangojs.po файл не включаючи генерованої шапки

```
1 #: static/js/main.js:71 static/js/main.js.c:92 static/js/main.js.c:1\  
2 15  
3 msgid "There was an error on the server. Please, try again later."  
4 msgstr "На сервері виникла помилка. Будь-ласка, спробуйте пізніше."
```

Дана стрічка зустрічється тричі у файлі main.js.

Після цього можна скомпілювати файли перекладу. Команда compilemessages залишається аналогічно до тієї, яку ми уже неодноразово використовували.

Оновлюємо сторінку (при потребі варто зробити один раз форс-релоад, щоб звільнити браузерні кеші) і переконуємось, що текст про помилку тепер відображається українською мовою.

...

Як бачите, процес інтернаціоналізації та перекладу стрічок в Javascript коді є дуже подібним до того, що ми робили в Python коді. Єдина відмінність - підготовка скрипта, що для нас генерує в'юшку javascript_catalog.

На даний момент ми з вами попрактикувались у всіх необхідних моментах при перекладі веб-аплікації. Але залишається ще один момент, щодо активації мови на веб-сайті. Давайте коротенько оглянемо як це працює і спробуємо переключати мову нашої аплікації базуючись на налаштуваннях відвідувача.

Активуємо мову під користувача

В теоретичній частині даної глави ми уже коротенько оглянули стратегії щодо активації тої чи іншої мови на сайті. І знаємо, що найбільш поширеними шляхами переключення мов є:

- вибір мови анонімним користувачем на публічному веб-сайті; це можуть бути прапорці країн або меню мов;
- вибір мови залогованим користувачем на веб-сайті у своїй панелі управління, персональних налаштуваннях;
- відображення веб-аплікації на мові вибраній відвідувачем у налаштуваннях свого веб-браузера.

На даний момент у нас мова веб-інтерфейсу є українська. Вона “прибита” в модулі налаштування проекту settings.py:

Дефолтна мова проекту

```
1 # default language
2 LANGUAGE_CODE = 'uk'
```

Тому кожен, хто завітає на наш веб-сайт, отримає стрічки графічного інтерфейсу українською мовою. Незалежно від персональних уподобань чи налаштувань браузера.

Django веб-фреймворк дає нам можливість однією стрічкою коду активувати механізм, яки включатиме ту мову сайту, яка обрана у налаштуваннях веб-браузера відвідувача.

Одна із дефолтних мідлвар (далі в наступних главах ми з вами детально ознайомимось, що таке middleware) Django вміє розпізнавати мовні налаштування браузера відвідувача сайту і динамічно, на кожному запиті, включати потрібну мову.

Для цього потрібно дану мідлвару додати у список активованих в проекті мідлвар:

Додаємо LocaleMiddleware до списку в налаштуваннях проекту, `settings.py`

```
1 MIDDLEWARE_CLASSES = (
2     'django.contrib.sessions.middleware.SessionMiddleware',
3     'django.middleware.locale.LocaleMiddleware',
4     'django.middleware.common.CommonMiddleware',
5     'django.middleware.csrf.CsrfViewMiddleware',
6     'django.contrib.auth.middleware.AuthenticationMiddleware',
```

На третьому рядочку коду бачимо нову компоненту LocaleMiddleware. Вона перевірятиме ‘Accept-Language’ заголовок, що приходить з браузера і активуватиме потрібну мову. Звісно, якщо дана мова є серед доступних перекладів у нашому проекті.

Зробіть необхідні зміни в модулі `settings.py` і потестуйте чи працює LocaleMiddleware клас. Для цього перезапустіть сервер Django і оновіть домашню сторінку нашої applікації.

Тепер зайдіть в налаштування вашого веб-браузера і поставте іншу мову (українську або англійську) першою в списку. Збережіть зміни і знову оновіть домашню сторінку. В результаті повинні отримати іншу мову на графічному інтерфейсі користувача.

Отак, однією стрічкою коду ми враховуємо побажання нашого відвідувача щодо зручної йому мови.

На самостійне опрацювання пропоную реалізувати вибір мови на інтерфейсі користувача. Щоб будь-який відвідувач даноїapplікації бачив меню з доступними мовами і міг зробити вибір. Вибір запам'ятується (для цього можна скористуватись кукою) і працює аж доки користувач не змінить мову. Організувати даний функціонал можна через в'юшку, яка братиме до уваги дану куку. А як переключати мови: гляньте у код мідвари 'django.middleware.locale.LocaleMiddleware'. Там знайдете необхідний модуль і функцію, яка вміє активувати вказану мову.

Django містить цілий міні-фреймворк для роботи із залогованими користувачами. Його ми розглянемо в наступних главах. Проте, я пропоную не чекати на детальні роз'яснення в книзі і спробувати самостійно розібратись із вбудованими можливостями по реєстрації та логуванню користувачів на сайті. Закрити анонімний доступ до нашої веб-аплікації. Розробити сторінку налаштувань залогованого користувача, в якій також помістити Мову. Дану мову визначатимемо мову інтерфейсу для даного користувача. Завдання доволі об'ємне і непросте. Але я обіцяю вам масу задоволення і пригод в процесі його виконання. Навіть, якщо не вдасться закінчити, все одно набудете досвіду та практики після одного повного дня роботи над даним завданням.

Домашнє завдання

От і підійшли ми до завершення даної глави. Надіюсь тепер ви непогано уявляєте, що таке інтернаціоналізація та локалізація програмного забезпечення та можете проробити мінімальний набір необхідних дій, щоб перекладач у вашому проекті зміг з легкістю адаптувати програму під різні мови.

Складність даної глави полягає лише у кількості нової теоретичної інформації для освоєння та розуміння. Але на практиці переклади це досить рутинне зав-

дання, що зводиться до перегляду усіх файлів та адаптації стрічок графічного інтерфейсу для подальшої локалізації.

Щоб закріпити даний матеріал даю ще кілька додаткових завдань окрім тих, що ми згадали під час глави.

...

В практичних цілях пропоную вам самостійно локалізувати всю аплікацію під ще одну мову. На ваш вибір. Переконайтесь, що перекладені усі стрічки. Це доволі просто зробити: достатньо додати опцію додаткової локалі в команду `makemessages`. Далі набити усі перекладти і скомпілювати кінцеві `“*.po”` файли. Якщо не володієте іншими мовами, то скористайтесь сервісом translate.google.com³⁷⁴.

Тепер переходимо до складніших завдань.

Якщо вам вдасться реалізувати форму налаштувань залогованого користувача, тоді додайте на цю форму також поле з вибором часової зони користувача. Врахуйте цю зону для відображення дат і часу у в'юшках та шаблонах. Таким чином, кожен користувач може мати власну обрану часову зону. Оскільки в нас поки в аплікації немає часу на інтерфейсі, то пропоную додати на форму редагування студента час останньої модифікації. Відповідно, відображати його у вибраній користувачем зоні.

Ще одне цікаве завдання. Спробуйте розібратись із вбудованою Django аплікацією `geoip`³⁷⁵. Вона дозволяє отримувати локацію користувача базуючись на його IP адресі. Ознайомтесь з її документацією і спробуйте вивести в шаблоні `base.html` місто (або країну) поточного відвідувача. Візуальну частину зробіть як вам до смаку. При цьому пам'ятайте: щоб IP відвідувача була інша, ніж `127.0.0.1`, вам потрібно заходити на вашу локальну аплікацію із іншої мережі. Або, альтернативно, закинути вашу аплікацію на інший сервер і туди самому заходити для тесту.

Ну і на завершення ще одне доволі непросте завдання. У цій главі ми займались лише перекладом графічного інтерфейсу аплікації. Але ми не торкалися

³⁷⁴<http://translate.google.com>

³⁷⁵<https://docs.djangoproject.com/en/1.7/ref/contrib/gis/geoip>

перекладу самого вмісту: студентів, груп, іспитів. Пропоную вам подумати над цим і спробувати реалізувати переклад студентів.

Для цього вам прийдеться обрати [одну із аплікацій](#)³⁷⁶ для перекладу моделей. Інтегрувати її із моделями студентів. Вивести на графічний інтерфейс органи управління перекладами студента. Таким чином перекладач зможе клонувати одного і того ж студента для різних мов і перекладати його поля.

Ось як це може виглядати:

Можливий інтерфейс перекладу об'єкта студента

Відповідно, коли матимете інтерфейс для перекладу об'єктів студентів онлайн, можна братись до оновлення сторінок. В списку студентів будуть відображатись лише ті копії студентів, які відповідатимуть за поточну активну мову.

...

В наступній главі ми з вами розглянемо дуже важливу тему: поняття залогованого користувача у веб-аплікації. Процеси логування, реєстрації та авторизації користувачів. Також запрограмуємо логування Facebook користувачів у нашу аплікацію. Практики буде значно більше, ніж у даній главі.

³⁷⁶<https://www.djangopackages.com/grids/g/model-translation>

Тому, як завжди, зробіть паузу після даної глави. Нехай ваша підсвідомість закладе теоретичний матеріал на потрібний рівень. Далі попрацюйте трохи над домашніми завданнями і лише тоді переходьте далі.

12. Доступ до аплікації: автентифікація користувачів

Однією із важливих тем при розробці веб-сайтів, та й загалом більшості інших програм, є тема користувачів. Більшість сучасних веб-аплікацій дозволяють анонімним відвідувачам ставати залогованими користувачами, які мають доступ до власних кабінетів та додаткових функцій.

Щоб освоїти дану теми ми базово розберемось із теорією:

- теорія користувачів, дозволів та ролей;
- методи логування у вебі;
- робота із користувачами у фреймворку Django.

Володіючи даною теорією ми зможемо справитись із практичними завданнями даної глави:

- інтегрувати систему користувачів і дозволити їм реєструватись та логуватись у нашу веб-аплікацію;
- “навішати” захист на усі сторінки та дії користувачів;
- розширити стандартну модель користувача Django додатковим полем;
- реалізувати сторінку профіля користувача;
- інтегрувати логін у веб-аплікацію через соцмережу Facebook.

Даний перелік завдань дозволить нам розібратись із основами роботи більшості систем користувачів у вебі. А почнемо ми із невеликого теоретичного вступу і коротенько оглянемо традиційну систему користувачів та методи логування користувача у вебі. Решту теорії будемо розбирати вже під час реалізації практичних завдань.

Теорія системи користувачів

Кожна програма, яка дозволяє заходити (логуватись) з власним іменем та паролем, обов'язково має організовану базу користувачів. Дані база користувачів в кожній із програм може бути організована по різному, а велика частина фреймворків та CMS систем надає можливість з легкістю підключати цілу множину популярних баз користувачів. Наприклад, таких як LDAP.

Таким чином, вперше потрапивши на новий для нас веб-сайт ми автоматично стаємо *анонімними* користувачами. Веб-сайт може надавати можливість зареєструватись, а тоді залогуватись щойно зареєстрованим користувачем. Залогувавшись, ми переходимо із режиму анонімного в режим *автентифікованого* (ще кажуть залогованого) користувача.

Кожен із типів користувачів, навіть анонімний, має свій набір *дозволів*. *Дозвіл* - це право на виконання однієї конкретної дії. Наприклад, переглядати сторінку контакту адміністратора, або право на реєстрацію (для аноніма).

Дозволи можуть напряму надаватись користувачеві, або через так звані *ролі*. *Роль* - це набір дозволів, які визначають політику поведінки певного типу користувача (аноніма, автентифікованого користувача, адміністратора і т.д.). В Django фреймворку немає поняття ролей, проте деякі інші веб-фреймворки та CMS системи використовують ролі для полегшення управління дозволами.

Також, щоб уникати “навішування” повторюваних ролей чи дозволів різним користувачам, більшість фреймворків реалізують поняття *групи користувачів*. *Групи користувачів* дозволяють об'єднувати користувачів за спільними ознаками та наданими їм можливостями. Після групування користувачів, групам надають дозволи або ролі і усі користувачі даної групи автоматично унаслідують дозволи групи.

Також більша частина фреймворків надає функціонал, що одразу дозволяє логувати, авторизувати та обмежувати доступ відповідно до прав користувача.

В різноманітних системах та фреймворках базова робота із користувачами може відрізнятись, проте вищеписана схема та поняття працюють для більшості сучасних популярних фреймворків.

Авторизація vs Автентифікація

Думаю, ви вже розумієте різницю між процесом логування (вхід існуючим користувачем) та процесом реєстрації (створення нового користувача).

Крім цього також маємо процес авторизації. Важливо відрізняти його від процесу автентифікації.

Як ми вже згадали раніше, процес *автентифікації* (authentication) - це отримання логіна та пароля від анонімного користувача, пошуку та співставлення із існуючим користувачем в базі. Якщо логін та пароль вірні, ми впускаємо користувача в систему: переводимо його у режим автентифікованого користувача.

Процес *авторизації* - це процес перевірки прав (дозволів) користувача і співставлення його із дозволами, необхідними для виконання потрібної користувачу дії. Якщо користувач не володіє необхідними правами, він отримає помилку про заборонений доступ або недостатні права.

Методи входу (логування)

Перед тим, як переходити до практичної частини варто згадати також про методи логування користувачів на сайті та поняття користувацьких сесій.

Розглянемо лише найбільш популярні методи:

Базова автентифікація (Basic)

HTTP протокол версії 1.0 визначає метод автентифікації Базовий. Він базується на імені користувача та його паролі. Якщо сервер отримує анонімний запит на захищений ресурс, то відправляє назад браузеру відповідь із статусним кодом 401 (Access Denied, Доступ Заборонено). Також до відповіді додається заголовок WWW-Authenticate:

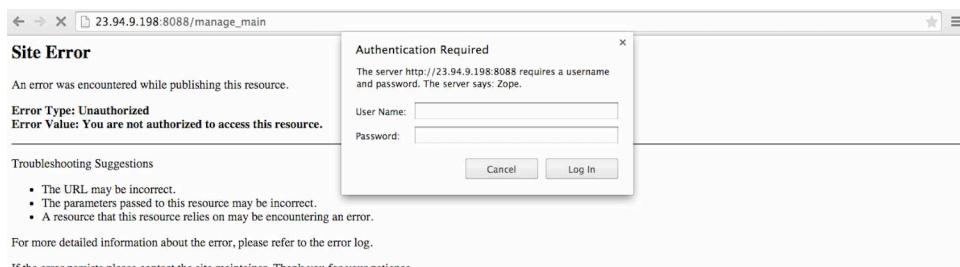
Заголовок WWW-Authenticate при Базовій автентифікації

1 HTTP/1.1 401 Access Denied

2 WWW-Authenticate: Basic realm="Test Server"

Заголовок WWW-Authenticate вказує на метод автентифікації (“Basic” у нашому випадку) та realm (“Test Server”). Стрічка realm може містити будь-яке значення. Вона зазвичай використовується веб-клієнтами, щоб керувати паролями на своїй стороні. Наприклад, щоб для різних сторінок дозволити окремо логуватись одним і тим самим браузером. Своєрідне групування паролів.

Більшість браузерів, отримавши заголовок WWW-Authenticate, відображають власне вікно логування, де користувач може ввести ім’я та пароль. Ця інформація далі відсилається на сервер і використовується для того, щоб ще раз спробувати отримати доступ до захищеного ресурсу.



Приклад браузерного вікна логування

Дані з вікна логування веб-браузер відправляє на сервер у заголовку Authorization:

Заголовок запиту Authorization

-
- 1 GET /protected-page/ HTTP/1.1
 - 2 Host: www.example.com
 - 3 Authorization: Basic aHR0cHdhdGNoOmY=
-

Знову ж таки, заголовок Authorization вказує на метод авторизації - Basic. А друга компонента містить ім’я та пароль користувача у формі: “<username>:<password>”. Дано стрічка не є шифрованою (криптованою), а лише закодованою з допомогою кодування base64³⁷⁷.

Саме тому даний метод логування вже давно не годиться для використання без додаткових інструментів криптування. Адже будь-хто зможе перехопити запит на сервер і отримати пароль користувача.

³⁷⁷ <http://uk.wikipedia.org/wiki/Base64>

Зауважте, закриптовані (агл. encrypted) і закодовані (англ. encoded) дані - це різні речі. Основне завдання *криптування* (шифрування) - приховати інформацію. Основне завдання *кодування* - привести дані у більш ефективний формат, для тої чи іншої цілі. Цілі можуть бути різні: зменшити кількість даних при передачі, економити енергію, забезпечити стійкість даних при помилках передачі. У вебі ми кодуємо дані в основному, щоб зменшити їхню кількість і економити на трафіку.

Дайджест автентифікація (Digest)

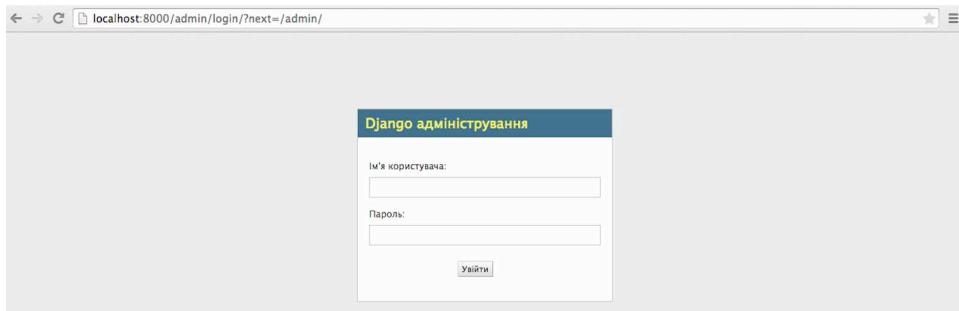
Дайджест автентифікація працює подібним чином до базової і також показує користувачу браузерне вікно логування. Єдиною відмінністю є те, що при дайджест автентифікації браузер надсилає криптований пароль на сервер. Таким чином, перехопити пароль в його оригінальному вигляді не вдасться.

А перехоплення його в хешованому вигляді також нічого не дасть, оскільки при дайджест автентифікації до пароля постійно додається довільно згенерована стрічка символів і вона змінюється при кожному запиті. Таким чином, перехоплений хеш вже не буде дійсний при наступному запиті.

Автентифікація з допомогою форми

Обидва попередні методи надають рідне браузерне вікно для логування. І розробник ніяк не може впливати на його вигляд та поведінку. Саме тому на сучасних веб-сайтах більшість форм логування - кастомна. Розроблена під потреби і вигляд веб-сайту.

Форма повинна містити два поля: ідентифікатор користувача (це може бути телефон, емейл, нікнейм, і т.д.) та пароль. Сервер приймає ці два поля і звіряється із базою користувачів, щоб дозволити чи відмовити у вході в систему.



Приклад кастомної форми логування з Django адмінки

Проте як і Базовий метод автентифікації, даний метод передає пароль користувача на сервер у абсолютно незахищенному вигляді. Тому в даному випадку нам необхідна ще одна додаткова технологія:

HTTPS протокол

Протокол HTTPS дозволяє шифрувати усі дані, що передаються від браузера до сервера. В тому числі ім'я та пароль користувача із форми логування.

Сам протокол базується на технології SSL³⁷⁸ (Secure Sockets Layer). Дані технологія використовує наперед приготовані приватний та публічні ключі в поєднанні з підписаним сертифікатом, щоб обмінюватись криптованими даними між сервером та клієнтом.

Даний протокол сповільнює роботу веб-сайту, адже криптування даних займає додатковий час. Тому рекомендується обслуговувати лише ті сторінки через протокол HTTPS, які містять конфіденційні дані (пароль користувача, дані кредитних карт і т.д.).



Так виглядає HTTPS адреса в дії

HTTPS сертифікати та ключі налаштовуються на стороні сервера і більшість популярних веб-серверів підтримують дану технологію.

Ми не розбиратимемо на практиці HTTPS/SSL і, якщо вам цікаво, то подальше знайомство із даною технологією вам залишається на самостійну роботу.

³⁷⁸<https://ru.wikipedia.org/wiki/SSL>

Децентралізована автентифікація

На даний момент кожен із нас, користувачів інтернету, є зареєстрованим у великому числі веб-ресурсів:

- соціальні мережі;
- онлайн емейл клієнти;
- онлайн сервіси управління файлами, відео, фотографіями;
- веб-форуми і дискусійні групи;
- спеціалізовані онлайн інструменти (для бухгалтерів, дизайнерів, програмістів і т.д.).

Через це придумали ще один метод логування на веб-сайти: логування на веб-сайт з допомогою існуючого користувача на іншому веб-сайті. Також цей спосіб називають децентралізованою автентифікацією.

При такому логуванні ви взагалі можете не мати користувача на поточному веб-сайті, або він буде створений для вас повністю автоматично. Натомість сайт надає вам можливість залогуватись використовуючи інший сервіс. Це може бути Google, Facebook, Twitter і інші.

На даний момент одними із найпопулярніших протоколів, що дозволяють реалізувати децентралізовану автентифікацію є: OpenID, OpenAuth, OAuth. Більшість соціальних мереж використовують протокол OAuth 2-ї версії.

Такий метод логування дозволяє нам, маючи користувача в одній системі, використовувати його для логування на інших. При цьому нам не потрібно придумувати нові паролі і заповнювати численні форми реєстрації.

Наприкінці даної глави ми з вами інтегруємо Facebook логування у нашу веб-аплікацію. Це буде хорошим прикладом децентралізованого логування.

...

В даній главі ми з вами на практиці реалізуємо метод авторизації з допомогою власної кастомної форми. Це дозволить створити гарну форму логування і застосувати потрібний нам вигляд.

В реальних applікаціях (веб-сайтах) до даного методу автентифікації також додають SSL протокол. Так, щоб усі сторінки із формами реєстрації та логування (а також запити на обробку форм, зазвичай це запити із методом POST) обслуговувались через HTTPS.

Таким чином автентифікація з допомогою власних форм поєднана з SSL криптуванням на даний момент є найпоширенішим методом логування. При цьому децентралізована автентифікація набирає все більших оборотів.

Сесії користувачів

Зареєструвати користувача і залогувати є лише половиною справи. Наступним непростим завданням веб-розробника є залишати користувача залогованим на усі наступні запити після того як він був автентифікований.

В протоколі HTTP, як ми вже знаємо, немає зв'язку між запитами. Кожен запит починає все спочатку: відкривається новий процес на сервері, запускається інтерпретатор мови, виконуються запити в базу, застосовується логіка, формується користувач, формується відповідь і віддається на клієнт (веб-браузер).

Саме тому нам, як веб-програмістам, приходиться обходити це обмеження HTTP протоколу. І вже давньою придумали поняття так званих *користувачьких сесій*. Як тільки користувач логується, сервер пробує запам'ятати цю його дію, щоб при кожному наступному запиті відновлювати запам'ятовані дані про залогованого користувача. Усі ці дані про факт логування і називаються користувачкою сесією.

Коли користувач логується, сервер створює нову сесію і присвоює їй унікальний ідентифікатор (часто його називають token англійською). Саме цей ідентифікатор і “гуляє” між запитами, щоб обійти обмеження протоколу HTTP. Щоб не передавати логін і пароль між запитами, придумали передавати саме ідентифікатор сесії.

Ще давніше ідентифікатор сесій ходив в адресах усіх запитів. Також його можна передавати через SSL механізм. Але в наш час найбільш популярним місцем збереження даного ідентифікатора - це браузерні куки. При логуванні кука з ідентифікатором сесії встановлюється на певний час, а після виходу користувача із сайту - стирається.

На сервері дані про користувацьку сесію можуть зберігатись в різноманітних місцях. По-замовчуванню Django зберігає користувацькі сесії в базі даних.

Альтернативно, оскільки користувацькі сесії - це тимчасові об'єкти, їх можна зберігати в кешах, у файловій системі, в оперативній пам'яті. Метод зберігання залежить від потреб проекту.

Тому знайте, як тільки ви логуєтесь у веб-сайт, для вас автоматично створюється сесія на сервері, а також додається хоча б одна нова кука до вашого веб-переглядача.

На даний момент цієї теорії є достатньо, щоб продовжувати далі. А почнемо ми із налаштування середовища для роботи із користувачами у нашій аплікації.

Налаштовуємо середовище користувачів

Веб-фреймворк Django уже надає нам механізм роботи з користувачами та їхніми сесіями. В Django ми маємо користувачів, групи та дозволи. Дозволи напряму “чіпляємо” користувачам та групам. Ролей в Django не існує.

Вбудована аплікація `django.contrib.auth`³⁷⁹ надає нам модель користувача, модель групи, дозволи, форми та логіку для логування, зміни паролю та виходу із веб-сайту. Вбудована аплікація `django.contrib.sessions`³⁸⁰ надає модель для зберігання користувацьких сесій в базі даних і, звичайно, усю логіку для управління ними.

Обидві аплікації повинні бути присутні серед списку заінстальованих в модулі налаштувань `settings.py`:

³⁷⁹ <http://djbook.ru/rel1.7/topics/auth/index.html>

³⁸⁰ <http://djbook.ru/rel1.7/topics/http/sessions.html>

INSTALLED_APPS в settings.py

```
1 INSTALLED_APPS = (
2     ...
3     'django.contrib.auth',
4     'django.contrib.sessions',
5     ...
6 )
```

Зазвичай, команда `startproject` при початковому створенні проекту уже додає ці дві аплікації у список `INSTALLED_APPS`.

Кожна із даних аплікацій приносить так звані “мідлвари” (middleware), які реалізують механізм сесій та користувача і роблять їх доступними в якості атрибутів на кожному запиті (`request`).

Ось список мідлвар, що повинні знаходитись в змінній `MIDDLEWARE_CLASSES` у вашому модулі `settings.py`:

MIDDLEWARE_CLASSES в модулі settings.py проекту

```
1 MIDDLEWARE_CLASSES = (
2     ...
3     'django.contrib.sessions.middleware.SessionMiddleware',
4     'django.contrib.auth.middleware.AuthenticationMiddleware',
5     'django.contrib.auth.middleware.SessionAuthenticationMiddleware',
6     ...
7 )
```

Давайте коротенько пройдемось по кожній з даних компонент:

- `SessionMiddleware`: додає підтримку користувацьких сесій і встановлює сесію користувача у кожен об'єкт запиту (доступний як `request.session`); зв'язок між запитами реалізований через браузерну куку;
- `AuthenticationMiddleware`: використовуючи користувацькі сесії присвоює кожному запиту на сервер об'єкт користувача (доступний як `request.user`);

- SessionAuthenticationMiddleware: дану компоненту додали в Django починаючи із версії 1.7; починаючи з версії 1.7.2 її функціонал перенесли у функцію `get_user` (яка використовується в мідлварі AuthenticationMiddleware) і залишили в списку MIDDLEWARE_CLASSES лише для того, щоб фреймворк включав верифікацію користувацьких сесій; тому, якщо ми хочемо, щоб сесії користувачів ставали недійсними, наприклад, після оновлення паролю, тоді повинні мати клас SessionAuthenticationMiddleware серед списку мідлвар; починаючи з версії 2.0 планується верифікацію сесій включати по-замовчуванню і тоді не буде потреби в даному класі взагалі.

Таким чином, переконайтесь що вищеописані класи і аплікації знаходяться у вас на потрібних місцях в модулі налаштувань `settings.py`. Після цього потрібно перестрахуватись і переконатись, що усі існуючі чи новододані аплікації синхронізовані із базою даних. Запустимо скрипт міграції:

Запускаємо міграційний скрипт

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb
2 $ python manage.py migrate
```

Тепер ми на всі сто готові продовжувати далі. А почнемо із інтеграції форми логування та механізму реєстрації нових користувачів у нашій аплікації.

Інтегруємо логін та реєстрацію

В Django уже є модель для об'єкта користувача - `User`. Дано модель містить такі поля як:

- ідентифікатор користувача;
- пароль;
- ім'я;
- прізвище;
- емейл адреса;
- активність (діючий користувач чи ні);

- статуси (персонал сайту чи звичайний користувач);
- список груп;
- список дозволів;
- дати приєднання та останнього входу на сайт.

Цього мінімуму вистачає для правильного функціонування користувачів. Але у більшості проектів потрібно розширяти набір полів користувача, щоб надати йому додаткову інформацію. Наприклад: адресу проживання, телефон і т.д. Цим ми і займемося трохи згодом у даній главі.

Також Django веб-фреймворк надає повний функціонал з автентифікації користувача. І після оформлення власної простенької форми логування ви одразу дозволите існуючим користувачам входити у сайт.

Для створення нового користувача в Django є лише адміністративна частина, що дозволяє створювати новий об'єкт моделі User. Для того ж, щоб дозволити анонімам самостійно реєструватись, потрібно самостійно реалізувати в'юшку та шаблони форм реєстрації.

Проте в Django спільноті існує [маса аплікацій](#)³⁸¹, які дозволяють повністю керувати користувачами, надають логін та реєстрацію, а також логін у веб-сайт через сторонні сервіси і соцмережі. Тому самостійно розробляти будь-які форми управління користувачами вже давно немає потреби. Хіба що вам потрібно задовільнити дуже специфічні вимоги проекту.

Серед популярних аплікацій, які рекомендую використовувати у своїх майбутніх проектах:

- [django-registration](#)³⁸²: надає реєстрацію користувачів на веб-сайті; тобто з допомогою даної аплікації вам не прийдеться самостійно писати код з реєстрації нових користувачів; в даній секції ми скористаємося даною аплікацією;
- [python-social-auth](#)³⁸³: колишня Django аплікація django-social-auth промігрувала у даний пакет і надає широкий спектр реєстрацій користувачів з допомогою зовнішніх веб-сервісів (Facebook, Twitter, Google+ і

³⁸¹<https://www.djangoproject.com/grids/g/registration/>

³⁸²<https://pypi.python.org/pypi/django-registration>

³⁸³<https://pypi.python.org/pypi/python-social-auth/>

т.д.); даною аплікацією ми скористаємось наприкінці даної глави, коли зайдемося інтеграцією Facebook логіну;

- **django-allauth³⁸⁴**: повністю інтегроване комплексне рішення, що надає і логування, і реєстрацію, і логін через сторонні сервіси.

Ми також не будемо власноруч писати код для реєстрації користувачів, а лише шаблон форми реєстрації. Для цього скористаємось аплікацією django-registration:

Інсталюємо django-registration

Першим ділом потрібно заінсталювати django-registration пакет, щоб далі ним скористатись. Але користуватись будемо не саме ним, а його гілкою під назвою **django-registration-redux³⁸⁵**, яку продовжують активно розробляти після того як головний контраб'ютор основної гілки зупинив свою роботу над аплікацією ще у 2013 році.

Зробимо це через requirements.txt файл:

Додаємо django-registration-redux до requirements.txt в корені репозиторію

```
1 django-registration-redux==1.2
```

На даний момент 1.2 - це остання версія гілки пакету. Ми готові до запуску команди інсталяції утиліти pip:

Інсталюємо нові пакети

```
1 # не забувайте активувати своє віртуальне середовище!
2 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb
3 $ pip install -r requirements.txt
```

Додаємо її до списку інсталюваних в проекті Django аплікацій:

³⁸⁴<https://pypi.python.org/pypi/django-allauth>

³⁸⁵<https://pypi.python.org/pypi/django-registration-redux>

settings.py

```
1 INSTALLED_APPS = (
2     ...
3     'registration',
4     'students',
5 )
```

Аплікація django-registration приносить нові моделі і тому для повної інсталяції треба прогнать ще раз міграцію:

Синхронізуємо базу даних із новою аплікацією django-registration

```
1 $ python manage.py migrate
2 Operations to perform:
3   Synchronize unmigrated apps: crispy_forms, registration
4   Apply all migrations: admin, students, contenttypes, auth, sessions
5 Synchronizing apps without migrations:
6   Creating tables...
7     Creating table registration_registrationprofile
8   Installing custom SQL...
9   Installing indexes...
10 Running migrations:
11   No migrations to apply.
```

На даному етапі ми маємо все необхідне, щоб інтегрувати форми логування існуючих та реєстрацію нових користувачів у нашу аплікацію.

...

Зробіть невелику паузу і самостійно складіть на папері свій список завдань, що дозволить відвідувачам реєструватись, логуватись і користуватись аплікацією в якості залогованих користувачів...

Подумали? А тепер порівняйте ваш список із цим:

- налаштовувати наш проект і аплікацію `django-registration`³⁸⁶, щоб дозволити реєстрацію користувачів;
- додати на усі наші сторінки лінки типу “Залогуватись” та “Зареєструватись”, щоб відвідувачі могли ними скористатись;
- подібним чином лінк “Вийти” також має бути видимий у випадку, якщо користувач уже автентифікований;
- заставити дані лінки працювати належним чином; для цього ми підготуємо власні шаблони та додамо нові URL заготовки в `urls.py` модуль;
- переглянути наші сторінки і в'юшки та захистити їх від анонімів.

Даний список буде нашим мінімальним набором завдань. А почнемо його із налаштування проєкту та `django-registration` аплікації:

Налаштовуємо `django-registration`

Єдиним налаштуванням аплікації `django-registration` є флагок `REGISTRATION_OPEN`³⁸⁷. Він вмикає чи вимикає реєстрацію нових користувачів на сайті. Ми, звичайно, повинні встановити його у `True`, щоб дозволити нові реєстрації. Дану змінну вказуємо в модулі `settings.py` нашого проєкту:

Активуємо реєстрацію нових користувачів

```
1 REGISTRATION_OPEN = True
```

На даний момент це єдине налаштування в модулі `settings.py`. Далі в секції ми ще неодноразово повернємося до даного модуля проєкту і по мірі потреби додаватимемо нові налаштування.

Спробуйте зануритись в код аплікації `django-registration` і знайти усі місця в коді, де використовується змінна `REGISTRATION_OPEN`. Зверніть увагу як дану змінну отримують та що з нею далі роблять у коді. Звичайно, ви не повністю зрозумієте всієї логіки копнувши лише частинку коду без загального контексту. Але ця вправа заставить вас ще раз почитати чужий

³⁸⁶<https://django-registration.readthedocs.org/en/latest/quickstart.html>

³⁸⁷<https://django-registration.readthedocs.org/en/latest/simple-backend.html>

якісний код, спробувати у ньому розібратись та запозичити нові техніки у власну практику.

Додаємо лінки до base.html

Першим ділом додамо в наш базовий шаблон лінки для логування та реєстрації. Коли користувач є анонімом, тоді він бачитиме лінки, щоб залогуватись та зареєструватись. А усі автентифіковані користувачі бачитимуть своє користувацьке ім'я та лінк для того, щоб вийти з аплікації.

Відкриваємо шаблон base.html, що знаходиться в аплікації students та оновлюємо шапку сторінки. Вставимо нові елементи на рівні меню груп, одразу за ним:

Додаємо користувацькі лінки в шапці шаблона base.html

```
1  <!-- Start Header -->
2  <div class="row" id="header">
3      <div class="col-xs-6">
4          <h1>{% trans "Students Accounting Service" %}</h1>
5      </div>
6
7      {% if user.is_authenticated %}
8          <div class="col-xs-4" id="group-selector">
9              <strong>{% trans "Group" %}:</strong>
10             <select>
11                 <option value="">{% trans "All Students" %}</option>
12                 {% for group in GROUPS %}
13                     <option value="{{ group.id }}"{% if group.selected %}selected="1"{% endif %}>{{ group.title }}{% if group.leader %}, {{ group.leader }}{% endif %}</option>
14                 {% endfor %}
15             </select>
16         </div>
```

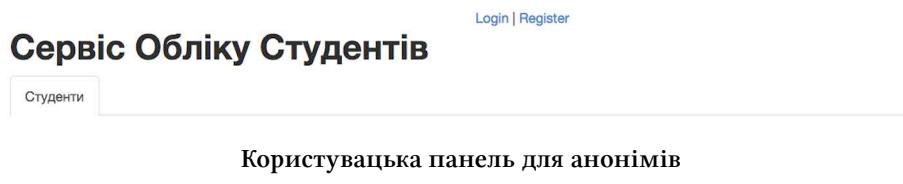
```
19      <!-- User's toolbar for authenticated users -->
20      <div class="col-xs-2" id="user-toolbar">
21          <span>{{ user.username }}</span> |
22          <a href="#">{% trans "Logout" %}</a>
23      </div>
24
25
26      {% else %}
27
28      <!-- User's toolbar for anonymous users -->
29      <div class="col-xs-6" id="user-toolbar">
30          <a href="#">{% trans "Login" %}</a> |
31          <a href="#">{% trans "Register" %}</a>
32      </div>
33      {% endif %}
34
35  </div>
36  <!-- End Header -->
```

Вище наведена лише частина шаблона `base.html`, що починається із хідера `Start Header`. Давайте детальніше оглянемо оновлені рядки:

- 4ий рядок: ми нічого не змінювали в текстовому лого нашої аплікації;
- 7ий: а от елемент меню груп ми огортаємо в умовний оператор `if`; якщо користувач залогований, лише тоді показуємо меню вибору груп, користувацьке ім'я та посилання Вийти; дефолтна Django аплікація `django.contrib.auth` підключає процесор контексту, який надає дві змінні у всі шаблони проекту: `user` (поточно залогований користувач або об'єкт анонімного користувача) і `perms` (дозволи поточну залогованого користувача); об'єкт користувача має метод `is_authenticated`, який дозволяє дізнатись чи користувач є залогованим;
- 8ий: саме меню груп ми також залишили без змін;
- 20ий: додали новий елемент одразу після меню груп, який показуємо також лише для залогованих користувачів - користувацьку панель;

- 22ий: для залогованих користувачів дана панель міститиме ім'я користувача; ім'я користувача беремо із об'єкта користувача user під атрибутом username; в подальших секціях даний текст буде лінком і вказуватиме на профіль користувача; після лінка ми поставили вертикальну лінію, щоб візуально розмежувати користувацьке ім'я та лінк для виходу із сайту;
- 23ий: також дана панель міститиме посилання Logout, яке дозволить користувачу знову стати анонімом; текст лінка огортаємо в тег перекладу; пізніше нам прийдеться локалізувати усі нові стрічки на користувацькому інтерфейсі; поки ми залишили адресу посилання із значенням решітки (#), адже у нас ще немає готових URL шаблонів під даний лінк; до них ми зовсім скоро дійдемо;
- 26ий: гілка else умовного оператору виводить варіант користувацької панелі для анонімів;
- 30ий: для анонімів ми показуватимемо лінк Login; його також динамізуємо пізніше, коли інтегруватимемо django-registration в'юшки на сервері;
- 31ий: також нам потрібно показувати лінк на форму реєстрації нових користувачів.

Ось як буде виглядати шапка вашої сторінки після вищенаведених змін:



Користувацька панель для анонімів

На даному зображені ви бачите лінки, що ведуть на форми логування та реєстрації. Як бачите, меню груп тепер пропало, адже ми не залоговані на сайті. Якщо ж ви бачите меню для залогованого користувача (тобто користувацьке ім'я та лінк Logout), вилогуйтесь із адміністративної частини Django.

Давайте також трохи підкоригуємо позицію користувацької панелі, щоб вона була на одній лінії з логотипом і вирівнялась по правий бік сторінки. Для цього потрібно відкрити файл із стилями main.css і додати два правила на наш новий елемент із ID 'user-toolbar':

students/static/css/main.css

```
1 /* User's Toolbar */
2 #user-toolbar {
3     margin-top: 30px;
4     text-align: right;
5 }
```

Правило margin-top опустило нашу панель вниз на 30 пікселів, а text-align заставило елементи панелі рівнятись по правій стороні контейнера (тобто панелі).

Зберігаєте зміни, оновлюєте сторінку (при потребі із форс-релоадом) і маєте бачити трохи кращу картинку:



Інтеграція на сервері

Картинку зробили, django-registration встановили. Залишається найцікавіше: динамізувати нашу картинку і інтегрувати логіку django-registration аплікації у наш проект.

Для цього нам потрібно буде:

- створити власні шаблони для форм логування та реєстрації;
- налаштувати в'юшки django-registration аплікації для роботи на потрібних нам адресах;
- оновити шаблон base.html працюючими адресами в панелі користувача.

Додаємо urls.py шаблони

А почнемо ми з додавання нових URL шаблонів у модулі urls.py проекту. Для цього додамо цілий параграф присвячений URL адресам під користувачів:

Початковий набір URL адрес для роботи з формами користувачів

```
1 ...
2 from django.contrib.auth import views as auth_views
3
4 ...
5
6 urlpatterns = patterns('',
7     ...
8     # User Related urls
9     url(r'^users/logout/$', auth_views.logout, kwargs={'next_page': \
10    'home'},
11         name='auth_logout'),
12    url(r'^register/complete/$', RedirectView.as_view(pattern_name='\
13 home'),
14        name='registration_complete'),
15    url(r'^users/', include('registration.backends.simple.urls',
16        namespace='users')),
17    ...
18 )
```

Спочатку ми імпортували в'юшки з дефолтної Django аплікації django.contrib.auth. При цьому перейменувавши локально даний модуль на auth_views, щоб запобігти конфлікту імен.

Найважливішим є останній регулярний вираз в рядку номер 15, який підключає усі шаблони з аплікації django-registration. А підключаємо їх у секцію URL адрес “users” із одноіменною назвою простору імен (namespace). Зверніть увагу, що ми звертаємось до набору URL шаблонів через ‘registration.backends.simple.urls’. В підпакеті simple лежить простіший варіант реєстрації користувачів - миттєвий. Після форми реєстрації людина одразу може входити на сайт.

Аплікація django-registration також надає варіант реєстрації з підтвердженням email адреси користувача. На домашнє завдання попрошу вас підключити і налаштувати даний варіант реєстрації. На більшості сучасних

вебсайтах задіяний саме такий варіант з емейл підтвердженням.

Модуль urls applікації django-registration перевіряє наявність змінної в налаштуваннях проекту INCLUDE_AUTH_URLS. Якщо дана змінна не є у False (а ми її взагалі не чіпали), тоді applікація підключає усі вбудовані в'юшки з модуля django.contrib.auth: login, logout і решту. Таким чином django-registration підключить їх для нас та ще й у область URL адрес: "/users/". Тепер маємо працюючі адреси users/login, users/logout, users/register і тому подібні. Єдине, що варто тут пам'ятати: дані в'юшки підключаються із іншими іменами URL шаблонів. django-registration додає до них префікси "auth_". Так, наприклад, URL шаблон під назвою login буде називатись auth_login.

З допомогою попередніх двох URL шаблонів (auth_logout і registration_complete) ми перекрили дефолтний функціонал django.contrib.auth та django-registration applікацій. Оригінальна в'юшка auth_logout (рядок 9ий у вищеприведеному прикладі) редіректить користувача на інформаційну сторінку про вихід із сайту. Натомість ми просто редіректимо його на головну сторінку. Робимо це з допомогою передачі додаткового ключа 'next_page' в аргументі kwargs функції url. А значенням даного ключа є назва URL шаблона, на який переводити користувача після дії Logout (в нашому випадку це home).

Подібним чином ми перекрили в'юшку registration_complete, щоб не показувати користувачу форму підтвердження реєстрації після заповнення форми. Натомість ми просто редіректимо його на головну сторінку після успішної реєстрації та логіну. Адже у спрощеному режимі реєстрації (режим simple в django-registration) користувачу не потрібно підтверджувати жодні дані і перевіряти поштові скриньки. А для редіректу ми скористалися вбудованою Django в'юшкою RedirectView передавши їй аргументом pattern_name назву URL шаблона куди редіректити. У нашому випадку це, знову ж таки, домашня сторінка веб-сайту.

Вам мабуть цікаво звідки я дізнався про те, які саме в'юшки потрібно перекривати? Є два варіанти: 1) почитати документацію обидвох applікацій

по роботі з користувачами; 2) зайти в модулі urls.py даних аплікацій і на власні очі перевірити, які URL шаблони вони надають. А вже методи перекривання в'юшок і URL адрес ми з вами розбирали у главі про в'юшки.

Динамізуємо посилання в панелі користувача

Коли маємо налаштовані нові адреси для логування, виходу, реєстрації та цілий ряд інших, можемо повернутись до шаблона base.html і динамізувати посилання в панелі користувача.

Знову відкриваємо base.html і оновлюємо лінки. Наводжу лише оновлені елементи:

Динамізуємо лінки в панелі користувача

```
1      ...
2
3      <!-- User's toolbar for authenticated users -->
4      <div class="col-xs-2" id="user-toolbar">
5          <span>{{ user.username }}</span> |
6          <a href="{% url "users:auth_logout" %}">{% trans "Logout" %}\ \
7      </a>
8      </div>
9
10     ...
11
12     <!-- User's toolbar for anonymous users -->
13     <div class="col-xs-6" id="user-toolbar">
14         <a href="{% url "users:auth_login" %}">{% trans "Login" %}<\/ \
15 a> |
16         <a href="{% url "users:registration_register" %}">{% trans "\ \
17 Register" %}</a>
18     </div>
19
20     ...
```

В рядку 6 можете бачити використання тега url, який використовує шаблон під назвою auth_logout. Даний шаблон зареєстрований в просторі шаблонів users, тому маємо викликати його через “users:auth_logout”. Подібним чином встановлюємо динамічні адреси для посилань Login та Register.

Щоб дізнатись усі доступні імена URL шаблонів, дивимось в документацію або код аплікації django-registration.

Оновіть сторінку в браузері і побачите, що наш лінк Login посилається на сторінку під адресою “<http://localhost:8000/users/login/>”.

А тепер спробуйте натиснути лінк Login і отримаєте “TemplateDoesNotExist at /users/login/: base.html”. А все тому, що django-registration аплікація надає усі шаблони, але вони працюють від base.html шаблона. Наш же ж базовий шаблон лежить під адресою students/base.html. Це можна легко виправити. Як? Це залишається вам на домашнє завдання описане нижче:

django-registration аплікація надає свої шаблони для усіх форм та сторінок під час роботи з користувачами. Але ми, для повторення матеріалу з формами, реалізуємо власні форми логування та реєстрації. На домашнє завдання вам залишиться розібратись із тим як скористатись усіма вбудованими формами django-registration аплікації. Налаштуйтесь на цікаву та не зовсім просту роботу з розбору документації та коду django-registration.

А ми з вами переходимо до створення власних шаблонів з формами логіну та реєстрації.

Реалізуємо форми логування та реєстрації

Постає наступне питання: куди покладемо усі шаблони та код, що пов’язаний із користувачами в нашій аплікації?

Нашу існуючу Django аплікацію students ми не будемо використовувати для цього. Адже вона містить моделі і логіку, що пов’язана із базою студентів,

груп та іспитів. В ідеалі уся логіка та шаблони по менеджменту системи користувачів повинні йти в окрему Django аплікацію. Проте ми не будемо повторювати рутинну роботу щодо створення та реєстрації нової Django аплікації. Натомість покладемо усі матеріали напряму в наш Django проект: `studensdb`.

На домашнє завдання вам треба буде створити нову аплікацію під назвою `stud_auth` і перенести туди усі шаблони, в'юшки, майбутні моделі та код пов'язаний із системою користувачів, яку ми інтегруємо в даній главі. Це буде хороша вправа на повторення роботи із аплікаціями. Це треба зробити подібним чином до того, як ми з вами створювали аплікацію `students` та інтегрували в Django проект `studensdb`.

Отже, спочатку створюємо папку `templates` в корені проекту `studensdb`. Туди ми будемо складати потрібні шаблони для користувачів. Також, щоб кастомізувати існуючі шаблони, які приходять з аплікації `django-registration`, потрібно класти їх у підпапку `registration` папки `templates`. Це ви можете побачити глянувши, знову ж таки, в документацію або код `django-registration`. Тому створюємо обидві папки одразу:

Створюємо папку `templates` в корені проекту

```
1 $ cd /data/work/virtualenvs/studensdb/src/studensdb
2 # опція -p створює усі вкладені папки в шляху,
3 # якщо їх ще немає
4 $ mkdir -p templates/registration
```

Тепер у нас є дві потрібні папочки. Проте папка `templates` в корені проекту автоматично не розпізнається фреймворком Django. Django розпізнає `templates` в аплікаціях, а `templates` в проекті потрібно власноруч додати до списку `TEMPLATE_DIRS` у модулі конфігурації `settings.py`:

Додаємо templates проекту до списку директорій із шаблонами

```
1 TEMPLATE_DIRS = (
2     os.path.join(BASE_DIR, 'studentsdb', 'templates'),
3 )
```

Змінна BASE_DIR уже визначена раніше у модулі settings.py. До кортежу TEMPLATE_DIRS ми додали єдиний поки елемент: шлях до папки templates в корені проекту studentsdb.

Не забудьте додати кому після елементу списку TEMPLATE_DIRS. В протилежному випадку Python інтерпретатор не сприйме змінну TEMPLATE_DIRS за кортеж, а просто за стрічку.

Усе. Ми готові до створення шаблона login.html всередині папки templates/registration. Саме так він повинен називатись, щоб django-registration аплікація його підхопила без додаткової конфігурації із нашої сторони:

Створюємо порожній файл login.html

```
1 $ touch templates/registration/login.html
```

Як альтернативний варіант можете скопіювати інший шаблон, що знаходиться в аплікації students. Адже нам знадобиться весь лейаут сторінки і унаслідування від базового шаблона base.html. Це зекономить вам трохи часу перенабирання тегів блоків.

Відкриваємо новостворений файл у редакторі і описуємо вручну повний HTML код форми і сторінки загалом:

Сторінка із формою логіну: registration/login.html

```
1  {% extends "students/base.html" %}  
2  
3  {% load i18n %}  
4  
5  {% block meta_title %}{% trans "Login Form" %}{% endblock meta_title%}  
6  
7  
8  {% block title %}{% trans "Login Form" %}{% endblock title %}  
9  
10 {% block content %}  
11  <form action="{% url "users:auth_login" %}" method="post">  
12  
13    {% csrf_token %}  
14    <input type="hidden" name="next" value="{% url "home" %}" />  
15  
16    <div class="form-group">  
17      <label for="login">{% trans "Your Username" %}</label>  
18      <input type="text" class="form-control" id="login" value=""  
19          name="username" />  
20    </div>  
21  
22    <div class="form-group">  
23      <label for="password">{% trans "Your Password" %}</label>  
24      <input type="password" class="form-control" id="password" value=\br/>25      ""  
26          name="password" />  
27    </div>  
28  
29    <button type="submit" class="btn btn-primary">{% trans "Login" %}<\br/>30  /button>  
31  
32  </form>  
33  
34  {% endblock content %}
```

Шапка шаблону у нас стандартна: ми унаслідуємось із базового шаблону, підключаемо інтернаціоналізацію, а далі вставляємо блоки. В деталі HTML коду занурюватись не будемо. Впевнений, він уже для вас повністю зрозумілий без пояснень. Тому оглянемо лише нові для нас елементи даного шаблону:

- 5ий рядок: заповнюємо блок мета заголовку текстом “Login Form”; пізніше нам потрібно буде його перекласти, а отже одразу не забуваємо про тег `trans`;
- 8ий: подібним чином набиваємо блок заголовка на сторінці;
- 10ий: починаючи з даного рядка ми заповнюємо основний блок сторінки `content`; він міститиме HTML код форми логування;
- 11ий: тег форми логування; атрибут `action` повинен вказувати на поточну сторінку; в'юшка `auth_login` працює у двох режимах взалежності від типу запиту (`get` або `post`): відображає форму та логує користувача; метод форми обов'язково повинен бути `post`;
- 14ий: приховане поле під назвою `next`, яке містить URL адресу сторінки, на яку редіректити користувача після вдалого логування; в'юшка логування бере до уваги даний параметр; у нашому випадку ми відправляемо користувача на головну сторінку;
- 16ий: починаючи з даного рядка ми вставляємо набір елементів для поля назви користувача; думаю додаткового пояснення дані елементи не потребують; подібним чином ми вставляємо поле для введення паролю.

Зверніть увагу, що дана форма не містить жодних елементів для розміщення повідомлень про помилки на формі. Це залишається вам на домашнє завдання: додати загальне статусне повідомлення і повідомлення біля кожного з полів. Коли виконаете, порівняйте свій результат із кодом, що йде із книгою. Він містить дану форму з елементами відображення повідомлень про помилки.

Сторінки реєстрації, логування та інших форм, які передають чутливі дані на сервер, повинні обслуговуватись через HTTPS протокол. Це збереже чутливі дані при перехопленні по дорозі на сервер. Ми не будемо налаштовували подібний захист у даній главі. Але він є обов'язковим на продакшін системах в реальних проектах.

Тепер клацніть по лінку Login і порівняйте вашу форму із даною:

The screenshot shows a web page titled "Сервіс Обліку Студентів". At the top right are links for "Login | Register". Below the title is a blue navigation bar with the text "Студенти". The main content area is titled "Login Form". It contains two input fields: "Your Username" and "Your Password", both with placeholder text. Below the password field is a "Login" button. At the bottom of the form is a copyright notice: "© 2014 Сервіс Обліку Студентів".

Форма логування готова

Спробуйте залогуватись скориставшись вашим існуючим користувачем. Це може також бути початковий адміністратор сайту, яким ви входите у адміністративну частину. Якщо з певних причин вас не редіректить на домашню сторінку у стані залогованого користувача (його можете ідентифікувати по зміненій панелі користувача), перегляньте спочатку усі кроки і переконайтесь, що все зробили як слід.

Таким чином, практично без Python коду ми змогли дозволити користувачам входити на сайт.

Знайдіть код, що відповідає за процедуру логіну користувача і спробуйте із ним поверхнево розібратись. Підказка: шукайте даний код в обробнику форми логування. А форма логування приходить із рідної Django аплікації

```
django.contrib.auth.
```

...

Подібним чином реалізуємо форму реєстрації. Щоб перекрити дефолтний шаблон аплікації django-registration, нам потрібно створити файл під назвою registration_form.html всередині registration папки.

Вам простіше буде скопіювати його із шаблону login.html. Після копіювання відкрийте у своєму редакторі і застосуйте необхідні зміни:

Вміст шаблона registration/registration_form.html

```
1  {% extends "students/base.html" %}  
2  
3  {% load i18n %}  
4  
5  {% block meta_title %}{% trans "Register Form" %}{% endblock meta_titl  
e %}  
6  
7  {% block title %}{% trans "Register Form" %}{% endblock title %}  
8  
9  {% block status_message %}  
10  {% if form.errors %}  
11    <div class="alert alert-warning" role="alert">{% trans "Please, corr  
12    ect the following errors." %} {{ form.non_field_errors }}</div>  
13  {% endif %}  
14  {% endblock %}  
15  
16  {% block content %}  
17    <form action="{% url "users:registration_register" %}" method="post">  
18      {% csrf_token %}  
19      <input type="hidden" name="next" value="{% url "home" %}" />
```

```
22 <div class="form-group">
23   <label for="login">{%
24     % trans "Your Username" %}</label>
25   <input type="text" class="form-control" id="login" value="" 
26     name="username" />
27   <span class="help-block">{{ form.errors.username.as_text }}</span>
28 </div>
29
30
31 <div class="form-group">
32   <label for="email">{%
33     % trans "Your Email" %}</label>
34   <input type="text" class="form-control" id="email" value="" 
35     name="email" />
36   <span class="help-block">{{ form.errors.email.as_text }}</span>
37 </div>
38
39 <div class="form-group">
40   <label for="password">{%
41     % trans "Your Password" %}</label>
42   <input type="password" class="form-control" id="password" value="" 
43     name="password1" />
44   <span class="help-block">{{ form.errors.password1.as_text }}</span>
45 </div>
46
47 <div class="form-group">
48   <label for="password-confirm">{%
49     % trans "Confirm Password" %}</label>
50   <input type="password" class="form-control" id="password-confirm" 
51     value="" 
52     name="password2" />
53   <span class="help-block">{{ form.errors.password2.as_text }}</span>
54 </div>
55
56
```

```
57   <button type="submit" class="btn btn-primary">{% trans "Register" %}</button>
58
59
60 </form>
61
62 {% endblock content %}
```

Даний шаблон дуже подібний до сторінки із формою логування. Тут ми маємо лише кілька відмінностей:

- заголовки сторінки звісно інші;
- форма вказує на іншу дію (action): в'юшку registration_register;
- маємо більше полів: ім'я користувача, емейл, пароль та підтвердження паролю;
- також дана форма містить повідомлення про помилки валідації; можете запозичити їх для форми логування, якщо ви пропустили попередню домашку.

Знову ж таки, в'юшка реєстрації django-registration аплікації зробить за нас усю брудну роботу. Переконайтесь, що ви вийшли із залогованого стану (лінк Logout повинен уже працювати) і переходіть на форму реєстрації. Заповніть дані і переконайтесь, що після відправки форми ви опинились на головній сторінці в залогованому під новоствореним користувачем стані.

Після того, як переведете ваші власні форми на форми аплікації django-registration, на вас чекає ще одне завдання. Зробіть так, щоб ці автогенеровані форми використовували Twitter Bootstrap HTML код. Тобто, вам потрібно буде заставити форми django-registration аплікації працювати через Crispy Forms аплікацію.

Також переконайтесь, що лінк Logout коректно працює після того як ви автентифікуєтесь на сайті.

В залогованому стані ви бачитимете ім'я свого користувача одразу перед посилання Logout. Ваше завдання: замість нього вставляти повне ім'я користувача, якщо поля Ім'я та Прізвище є непорожніми. Інакше й надалі вставляти його нікнейм (username).

...

Таким чином, малою кров'ю ми дозволили відвідувачам входити на сайт та реєструвати нових користувачів. Але поки автентифікований користувач бачить практично все те ж саме, що і анонім. Давайте заховаемо частину функціоналу від аноніма. Зазвичай, анонім може лише переглядати інформацію на веб-сайті, але не може вносити жодних змін в базу.

Тепер, коли маємо інтегровані логін та реєстрацію користувачів, саме час закомітити ваші зміни в репозиторій. Хоча, коміти можна було в даній секції робити і частіше: один після інсталляції і конфігурації django-registration, один після реалізації логіну, один після реалізації реєстрації і т.д.

Ховаємо функціонал від анонімів

Тепер маючи два типи користувачів на сайті - аноніми і залоговані користувачі, можемо визначитись, що кожен із типів може робити:

- **анонім** може бачити лише сторінку із списком студентів і не може виконувати жодні інші дії над студентами;
- **анонім** може реєструвати користувача та логуватись на сайті;
- **залогований користувач** може виконувати будь-які існуючі дії на інтерфейсі нашої веб-аплікації;
- проте відвідувати сторінку контакту адміністратора може лише користувач із дозволом “Add user”.

Додавання власного дозволу, що не є прив'язаним до моделі є нетривіальною задачею в Django. І зазвичай виконується з допомогою інсталяції додаткових аплікацій. Тому в даній секції ми “прив'яжемо” існуючий дозвіл “Add user” до закладки контакту адміністратора. Лише ті, хто його матимуть, зможуть відвідувати дану форму. Проте ви можете спробувати самостійно розібратись із кастомними дозволами, що не прив'язані до моделей. Якщо вдасться додати новий дозвіл, тоді спробуйте задіяти його для закладки контакту адміністратора.

Дане завдання є доволі об'ємним, оскільки заставить нас оглянути буквально усі в'юшки і URL шаблони нашого проекту. Саме тому воно заслуговує окремої секції.

Також дане завдання поділяється на дві великі частини:

- по-перше, нам потрібно буде обмежити певні сторінки, в'юшки і URL адреси на доступ для певної категорії користувачів; тобто ці користувачі бачитимуть попередження або помилку, або форму логування при спробі потрапити на недозволений для них ресурс;
- по-друге, краще недозволені ресурси взагалі не виводити на інтерфейсі користувача; наприклад: якщо користувач не має права відвідувати закладки контакту адміністратора, тоді взагалі її краще не показувати на сторінці.

Налаштовуємо адресу форми логіну

У випадку, якщо користувач є анонімом і пробує потрапити на ресурс, який є доступний лише для автентифікованих користувачів, Django редіректить його на сторінку із формою логування. Проте дефолтна адреса даної сторінки у фреймворку Django вказує на ‘/accounts/login/’. Наша ж форма логування знаходитьться тепер під адресою ‘/users/login/’.

Ми можемо перебити дане налаштування з допомогою змінної LOGIN_URL всередині модуля налаштувань settings.py проекту:

LOGIN_URL в модулі settings.py

```
1 LOGIN_URL = 'users:auth_login'  
2 LOGOUT_URL = 'users:auth_logout'
```

Ми присвоїли змінній LOGIN_URL назву URL шаблона, під яким обслуговуємо сторінку логіну у нашому проекті. Подібним чином працює і змінна LOGOUT_URL. Її ми також оновили, щоб була актуальною в контексті нашого проекту.

Захищаємо в'юшки роботи із студентами

Почнемо ми із огляду в'юшок, що працюють із студентами. Ось що нам прийдеться зробити недоступним для аноніма:

- додавання нового студента;
- форму редагування існуючого студента;
- видалення існуючого студента.

Сам список студентів можуть переглядати навіть аноніми, тому його захищати не потрібно.

Відкриваємо модуль views/students.py всередині нашої applікації та додаємо два нові імпорти:

Імпортуюмо додаткові функції

```
1 ...  
2 from django.utils.decorators import method_decorator  
3 from django.contrib.auth.decorators import login_required  
4 ...
```

Функція login_required використовується в якості декоратора на функції в'юшки. Після декорування в'юшки доступ до неї автоматично надається лише автентифікованим користувачам. Усі аноніми будуть переведені на адресу settings.LOGIN_URL, яку ми вже з вами налаштували.

Функція декоратор method_decorator пригодиться нам для використання декоратора login_required для захисту в'юшок, що є класами. Адже login_required початково призначена лише для декорування функцій, а не методів класу.

Продовжуємо роботу з модулем students.py і тепер захистимо в'юшку додавання нового студента від анонімів:

```
1 ...
2
3 @login_required
4 def students_add(request):
5     ...
```

Ось так одним додатковим рядком перед функцією students_add ми перекрили доступ до даної в'юшки. login_required приймає ряд додаткових параметрів, але тут вони нам не знадобляться. Спробуйте тепер анонімом потрапити на форму додавання нового студента. Якщо все зробили правильно, тоді замість форми додавання маєте отримати форму логування.

Рухаємось далі. На черзі в'юшки редагування та видалення студента. Дані в'юшки є класами і щоб захистити їх, нам прийдеться написати децьо більше рядочків коду:

Захищаємо процедури редагування та видалення студента

```
1 ...
2 class StudentUpdateView(UpdateView):
3     ...
4     @method_decorator(login_required)
5     def dispatch(self, *args, **kwargs):
6         return super(StudentUpdateView, self).dispatch(*args, **kwargs)
7
8
9 ...
10
11 class StudentDeleteView(DeleteView):
```

```
12     ...
13     @method_decorator(login_required)
14     def dispatch(self, *args, **kwargs):
15         return super(StudentDeleteView, self).dispatch(*args, **kwar\
16 gs)
17     ...
```

Як ви вже знаєте, стандартні Django в'юшки-класи мають метод `dispatch`. Через нього входять усі запити на в'юшку, а вже він вирішує, куди далі передавати даний запит. Тому, щоб перекрити загальний доступ до такої в'юшки, потрібно перекрити доступ саме до методу `dispatch`.

Це ми зробимо, знову ж таки, з допомогою декоратора `login_required`. Але тут ми маємо огорнути його в декоратор `method_decorator`, щоб перетворити `login_required` із декоратора функцій у декоратор методів.

Сам же ж метод `dispatch` ми додаємо до обидвох в'юшок `StudentUpdateView` та `StudentDeleteView`. Ми просто дублюємо усі аргументи даного методу, щоб один в один кастомізувати його у наших класах. Код методу простий: запускаємо усю логіку з батьківського класу передаючи усі отримані аргументи. При цьому ми скористалися функцією `super`.

Щоб не повторювати кастомізацію методу `dispatch` у всіх необхідних класах в'юшках, можна написати абстрактний клас, який міститиме оцей єдиний захищений метод `dispatch`. А тоді просто унаслідуватись від даного класу у всіх в'юшках, що потребують захисту від анонімів. Спробуйте цю вправу виконати самостійно.

Також на домашнє завдання пропоную вам подібним чином захистити від анонімів в'юшки груп, іспитів та журналу. Далі в даній секції ми захистимо їх іншим методом. А ви спробуйте пройтись по коду проекту та аплікації

по черзі застосувавши обидва методи захисту в'юшок.

Тепер перевірте чи ваш захист працює і чи ви дійсно отримуєте форму логування замість потрібних сторінок.

Захищаємо форму контакту адміністратора

На черзі у нас робота із формою контакту адміністратора. Тут ми перейдемо на більш детальний рівень і захистимо дану форму конкретним дозволом: “Can add user”.

Дозвола у фреймворку Django створюються через прив'язку до моделей. Якщо потрібно дозвіл, що немає відношення прямого до жодної із моделей, тоді треба придумувати обхідні шляхи або користуватись існуючими сторонніми аплікаціями. Але цього матеріалу ми не охоплюємо в даній главі.

По-замовчуванню, аплікація `django.contrib.auth` автоматично генерує дозвола для кожної зареєстрованої моделі: `add` (додати), `change` (змінити), `delete` (видалити). Вони створюються під час команди `migrate`, коли фреймворк ідентифікував ваші нові моделі.

Таким чином, для нашої аплікації `students` і моделі `Student`, ми автоматично отримали наступні дозволи в системі:

- `students.add_student`: для створення нового студента;
- `students.change_student`: для оновлення існуючого студента;
- `students.delete_student`: для видалення існюючого студента.

Сторінку контакту адміністратора ми прив'яжемо до дозволу `auth.add_user`. Тобто будь-який користувач, який має право створювати нових користувачів на сайті, зможе контактувати адміна.

Відкриваємо модель із в'юшкою `students/views/contact_admin.py`:

Захищаємо в'юшку контакту адміна дозволом

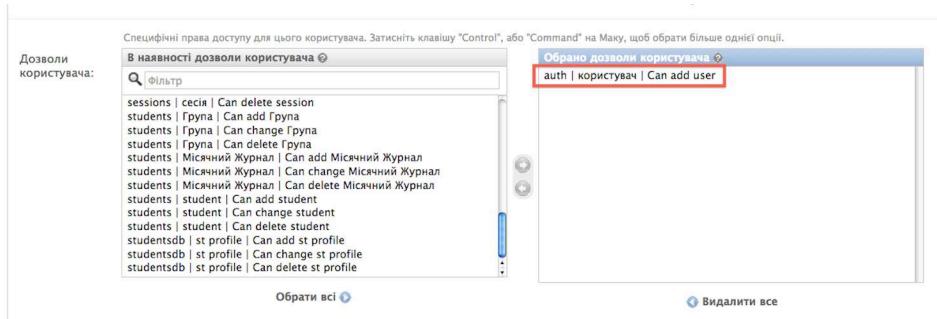
```

1 ...
2 from django.contrib.auth.decorators import permission_required
3 ...
4 ...
5 ...
6 @permission_required('auth.add_user')
7 def contact_admin(request):
8 ...

```

Спочатку ми імпортували нову функцію-декоратор `premission_required`. Дана функція отримує аргументом назву дозволу для захисту. Вона працює аналогічним чином як працює декоратор `login_required`. Після імпорту нам залишилось декорувати функцію-в'юшку `contact_admin` передаючи декоратору `permission_required` назву дозвола `auth.add_user`.

Це все, що потрібно нам для захисту форми контакту. Щоб потестувати даний захист, нам треба підготувати відповідного користувача. Можете створити нового або оновити існуючого. У адміністративній частині Django зайдіть на форму редагування обраного вами користувача і додайте дозвіл “Can add user”:



Додаємо дозвіл на формі редагування користувача

Збережіть форму і спробуйте залогуватись саме під цим користувачем на сайт. Якщо все зробили правильно, ваш користувач зможе зайти на закладку Контакт і не отримає форми логування, а побачить форму контакту адміністратора.

Контролюємо доступ з допомогою URL шаблонів

До цього часу ми захищали в'юшки з допомогою функцій-декораторів напряму оновлюючи їхній код. Проте існує і інший спосіб захисту: захист на рівні URL шаблонів.

Використовуючи ті ж самі функції для захисту, якими ми користувались у попередній секції, ми можемо захищати в'юшки прямо під час декларації URL шаблонів.

Як приклад, розберемо випадок із захистом набору URL адрес для роботи із групами:

Захищаємо сторінки роботи із групами через urls.py

```
1 ...
2 from django.contrib.auth.decorators import login_required
3 ...
4 from students.views.groups import GroupAddView, GroupUpdateView, \
5     GroupDeleteView, groups_list
6 ...
7 urlpatterns = patterns('',
8     ...
9     # Groups urls
10    url(r'^groups/$', login_required(groups_list), name='groups'),
11    url(r'^groups/add/$', login_required(GroupAddView.as_view()),
12        name='groups_add'),
13    url(r'^groups/(?P<pk>\d+)/edit/$',
14        login_required(GroupUpdateView.as_view()),
15        name='groups_edit'),
16    url(r'^groups/(?P<pk>\d+)/delete/$',
17        login_required(GroupDeleteView.as_view()),
18        name='groups_delete')
19    ...
20 )
21 ...
```

Як бачите, ми просто огорнули кожну в'юшку (результат виклику методу `as_view()`) у функцію `login_required`. Подібним чином можна працювати і з

функцією `permission_required`.

Ще одна додаткова річ, яку нам прийшло змінити, це замінити стрічкове посилання на в'юшку `groups_list`. Адже функція `login_required` приймає лише функцію і не вміє працювати із стрічками, що є шляхами до даних функцій.

На домашнє завдання попрошу вас оглянути усі URL шаблони в модулі `urls.py` і додати необхідний захист з допомогою декораторів `login_required` та `permission_required`.

Знову ж таки, потестуйте на сайті під анонімом чи спрацювали наші зміни в модулі `urls.py`.

Ховаемо недоступні сторінки з UI

На даний момент ви, будучи анонімом, отримуватимете форму логування при кліку по більшості посилань і кнопок на сторінці нашої аплікації. Погодьтесь, що такий сценарій дещо розчарує потенційного користувача. Саме тому рекомендується просто не показувати тих елементів навігації, які є недоступні поточному користувачеві.

В окремих випадках корисно показати попередження про необхідність залогуватись. Але ми просто заховамо усі елементи, до яких користувач не матиме доступу.

Щоб продемонструвати загальну ідею та інструментарій на рівні шаблонів, розберемо випадок із нашим головним меню. Ось що нам потрібно зробити:

- приховати закладки Групи, Іспити, Відвідування від анонімів;
- приховати закладку Контакт від усіх користувачів, окрім тих, які мають дозвіл `auth.add_user`.

Відкриваємо шаблон `base.html` і реалізуємо поставлені задачі:

Ховаемо закладки в base.html

```
1 ...
2     <!-- Start subheader -->
3     <div class="row" id="sub-header">
4         <div class="col-xs-12">
5             <ul class="nav nav-tabs" role="tablist">
6                 <li role="presentation" {% if request.path == '/' %}class=\
7 "active"{% endif %}><a href="{% url "home" %}">{% trans "Students" %}\
8             </a></li>
9
10            {% if user.is_authenticated %}
11                <li role="presentation" {% if '/journal' in request.path %}\
12                class="active"{% endif %}><a href="/journal">{% trans "Journal" %}<\
13 /a></li>
14
15                <li role="presentation" {% if '/groups' in request.path %}\\
16                class="active"{% endif %}><a href="{% url "groups" %}">{% trans "Gro\
17 ups" %}</a></li>
18
19            {% endif %}
20
21            {% if perms.auth.add_user %}
22                <li role="presentation" {% if '/contact-admin' in request.\
23 path %}class="active"{% endif %}><a href="{% url "contact_admin" %}"\>{% trans "Contact" %}</a></li>
24
25            {% endif %}
26
27        </ul>
28    </div>
29    </div>
30     <!-- End subheader -->
31 ...
32 ...
```

В 10 рядку ми обгорнули закладки Відвідування та Групи в умовний оператор. І показуємо їх лише для автентифікованих користувачів.

В рядку 19 ми обгорнули закладку Контакт у інший умовний оператор. Аплікація django.contrib.auth надає нам ще одну змінну в шаблонах: perms.

Дана змінна містить набір дозволів, якими володіє поточний користувач. Дані дозволи розподіляються по назвах моделей, до яких належать. Таким чином, щоб перевірити чи користувач володіє дозволом auth.add_user, нам потрібно дістати зі змінної perms атрибут auth, а вже з нього дозвіл add_user.

Збережіть файл, перевантажте сторінку у браузері і переконайтесь, що закладки працюють правильно для аноніма, залогованого користувача і користувача із дозволом auth.add_user.

На домашнє завдання вам залишається заховати меню Дії та кнопку додавання студента для анонімів. Також можете показувати ім'я та прізвище студента у списку не як посилання, а як звичайний текст. Адже анонім немає доступу до форм редагування студентів.

...

На цьому ми завершуємо короткий огляд Django інструментів для захисту окремих URL адрес та в'юшок. Цього вам буде достатньо для більшості випадків у реальних проектах.

А наступним пунктом на порядку денному у нас стоїть робота із розширенням моделі користувача додатковими полями та реалізація сторінки профіля користувача.

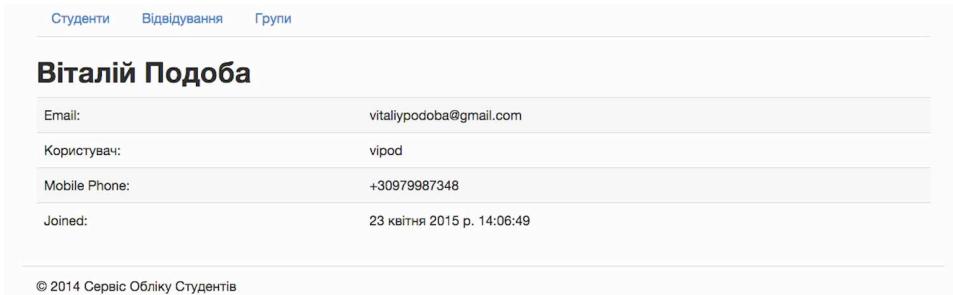
Розробляємо сторінку профіля

В даній секції ми не лише створимо сторінку профіля користувача, але й розширимо профіль одним додатковим полем. В більшості реальних веб-проектів перед нами постає потреба розширити дані користувача додатковими полями, що є специфічними для даного проекту. Тому розуміння варіантів як це можна зробити всередині фреймворку Django є доволі важливим.

А почнемо ми із простішого завдання. Створимо сторінку профіля зареєстрованого користувача.

Профіль користувача

Перед нами стоїть задача розробити наступну сторінку профіля користувача:



Сторінка користувача

Поле телефону ми поки пропустимо і зробимо на етапі, коли розшириємо профіль користувача додатковим полем.

Загальна ідея така, що залогований користувач бачить своє ім'я одразу перед посилюнням Logout. Дане ім'я ми залінкуємо на сторінку профіля, щоб користувач міг переглянути свої деталі.

Почнемо із додавання нового URL шаблону для адрес користувачьких профілів. Нехай адреса у форматі “/users/profile” показуватиме залогованого користувача. Тобто користувач може бачити лише свій профіль:

urls.py

```
1 ...
2 urlpatterns = patterns('',
3     ...
4     # User Related urls
5     url(r'^users/profile/$', login_required(TemplateView.as_view(
6         template_name='registration/profile.html')), name='profile'),
7     url(r'^users/logout/$', auth_views.logout, kwargs={'next_page': \
8         'home'},
9         name='auth_logout'),
10    url(r'^register/complete/$', RedirectView.as_view(pattern_name='\
11        home'),
```

```
12         name='registration_complete'),
13     url(r'^users/', include('registration.backends.simple.urls',
14                             namespace='users')),
15     ...
16 )
17 ...
```

Ми додали новий шаблон першим у групі шаблонів щодо користувачів. Насправді, у даному випадку порядок немає значення, адже усі дані шаблони не перетинаються. Єдиний шаблон, для якого важливий порядок - це підключення цілого ряду шаблонів через `include` під шляхом “/users/”. Його краще поставити наприкінці шаблонів користувачів, щоб уникнути випадкових конфліктів і перекритих адрес.

Наша сторінка профілю не вимагає кастомної Python в'юшки з логікою, адже все, що нам необхідно це доступ до об'єкта користувача. Відповідно, ми скористалися вбудованою Django в'юшкою `TemplateView`.

Наступним кроком підготуємо сам шаблон `profile.html` всередині папки `templates / registration`. Можете скопіювати його із шаблону `login.html`, щоб потім менше повторювати стандартного коду із блоками.

Ось як виглядатиме код сторінки профіля користувача:

profile.html

```
1  {% extends "students/base.html" %} 
2
3  {% load i18n %} 
4
5  {% block meta_title %}{{ user.get_full_name|default:user.username }}\ 
6  {% endblock meta_title %} 
7
8  {% block title %}{{ user.get_full_name|default:user.username }}{% en\ 
9  dblock title %} 
10
11 {% block content %}
```

```
13 <table class="table table-striped">
14     {% if user.email %}
15         <tr>
16             <td>{{ trans "Email" }}:</td>
17             <td>{{ user.email }}</td>
18         </tr>
19     {% endif %}
20
21     {% if user.username %}
22         <tr>
23             <td>{{ trans "Username" }}:</td>
24             <td>{{ user.username }}</td>
25         </tr>
26     {% endif %}
27
28     {% if user.date_joined %}
29         <tr>
30             <td>{{ trans "Joined" }}:</td>
31             <td>{{ user.date_joined }}</td>
32         </tr>
33     {% endif %}
34
35     </table>
36
37     {% endblock content %}
```

Давайте детальніше пройдемось по важливих рядках даного шаблону:

- 5ий: в мета заголовок ми вставляємо повне ім'я користувача використовуючи get_full_name метод об'єкту User; якщо поля Ім'я та Прізвище не заповнені, тоді отримаємо порожню стрічку; в такому випадку ми через фільтр default вставляємо користувацьке ім'я, яке є завжди присутнім;
- 14ий: якщо користувач заповнив свій емейл, вставляємо його у таблицю;
- 28ий: також об'єкт User має поле date_joined; це дата реєстрації користувача.

Сторінка користувача вийшла доволі простою. Зазвичай в реальних проектах на неї поміщають значно більше інформації та елементів: форма контакту, фотографія, кнопка потоваришувати і т.д. Але у нашому випадку ми обмежимось простим профілем. Не забудьте зберегти файл.

Останнім кроком перед тестом нашого профіля маємо динамізувати користувачьке ім'я всередині панелі користувача. Тому знову відкриваємо шаблон base.html, що знаходиться в аплікації students:

Динамізуємо користувачьке ім'я в панелі

```
1      <!-- User's toolbar for anonymous users -->
2      <div class="col-xs-2" id="user-toolbar">
3          <a href="{% url "profile" %}">{{ user.get_full_name|default:\
4              user.username }}</a> |
5          <a href="{% url "users:auth_logout" %}">% trans "Logout" %\n
6      </a>
7      </div>
```

Тег “span” ми замінили тегом “a”. Для генерації адреси на профіль користувача ми скористались тегом url, URL шаблоном profile та ідентифікатором користувача в якості аргумента. Таким чином, даний лінк працюватиме в контексті залогованого користувача. Текст лінка ми також оновили і тепер він буде показувати повне ім'я користувача, якщо таке присутнє.

Все. Ми готові до тесту. Оновлюйте сторінку у браузері. Залогуйтесь. Клікайте на ваше ім'я і переконайтесь, що сторінка є подібною на ту, яку ви бачили на початку даної секції.

На домашнє завдання вам потрібно буде покращити сторінку профіля так, щоб на неї могли заходити інші користувачі. Для цього треба буде реалізувати власну в'юшку і URL шаблон, що передаватиме в'юшці ID користувача. Наприклад, ‘/users/profile/<User ID>’ буде непоганим варіантом для профілів, до яких можна доступитись по ідентифікатору користувача.

Додаємо нове поле до профіля користувача

Як вже було сказано раніше, в реальних проектах практично завжди потрібно наділяти користувачів додатковими деталями. Як це робити саме в Django ми тут якраз з вами і розглянемо.

В Django фреймворку є щонайменше три шляхи “загачення” стандартної моделі користувача:

- якщо зміни до моделі User мають бути чисто функціональні, а не набивка додатковими даними, тоді можна обійтись так званими [проксі моделями](#)³⁸⁸;
- якщо ж вам потрібно зберігати в базі додаткові деталі про користувача, тоді можна використовувати додаткову модель зв’язавши її з моделлю User полем зв’язку OneToOneField; таку додаткову модель часто називають моделлю профіля і, зазвичай, вона зберігає деталі про користувача, які не мають прямого відношення до процесу автентифікації та авторизації в системі;
- проте в деяких проектах можуть бути особливі вимоги до моделі User; наприклад, Django використовує username поле для автентифікації користувача, але більшість реальних сайтів працює через емейл адресу; щоб змінити даний підхід, вам потрібно буде [повністю замінити модель User власною моделлю](#)³⁸⁹; цей процес є доволі непростим і вимагає міграції на самому початку проекту.

Для наших цілей (додати одне нове поле до деталей користувача) ми скористаємося саме другим підходом. Усі інші методи вам залишаються на самостійне ознайомлення і знаходяться поза межами даної книги.

Реєструємо проект як аплікацію

Другий метод розширення даних про користувача вимагає реалізації додаткової моделі. Цю модель, так само як і інші матеріали щодо роботи користувачів, ми покладемо у проект.

³⁸⁸<http://djbook.ru/rel1.7/topics/db/models.html#proxy-models>

³⁸⁹<http://djbook.ru/rel1.7/topics/auth/customizing.html#substituting-a-custom-user-model>

Django фреймворк розпізнає моделі лише у тих аплікаціях, які додані до списку INSTALLED_APPS в конфігурації проекту. Оскільки ми покладемо наш новий клас моделі в проект, тоді його також потрібно тепер додати до списку інсталльованих аплікацій.

Якщо ви уже виконали домашнє завдання і перенесли увесь код стосовно систему користувачів в окрему аплікацію, тоді модель профіля користувача ви також покладете у цю іншу аплікацію. Таким чином, ви можете пропустити наступні кроки у даній секції і переходити одразу до реалізації моделі профіля.

Відкриваємо знову settings.py модуль і додаємо studentsdb в кінець списку INSTALLED_APPS:

Реєструємо проект studentsdb як аплікацію

```
1 INSTALLED_APPS = (
2     ...
3     'registration',
4     'students',
5     'studentsdb',
6 )
```

Додаємо поле для телефону

Тепер можемо переходити безпосередньо до класу моделі. Назовемо її StProfile (скорочено від Students Profile) і помістимо її в модуль models.py в корені проекту studentsdb. Окремого підпакету для моделей створювати не будемо, адже ми не плануємо додавати більше моделей у код проекту.

Створіть порожній файл models.py в корені проекту і відкрийте його в редакторі. Нарешті попишемо трохи більше Python коду:

Реалізуємо клас моделі профіля

```
1 from django.db import models
2 from django.contrib.auth.models import User
3 from django.utils.translation import ugettext as _
4
5
6 class StProfile(models.Model):
7     """To keep extra user data"""
8     # user mapping
9     user = models.OneToOneField(User)
10
11     class Meta(object):
12         verbose_name = _(u"User Profile")
13
14     # extra user data
15     mobile_phone = models.CharField(
16         max_length=12,
17         blank=True,
18         verbose_name=_(u"Mobile Phone"),
19         default=' ')
20
21     def __unicode__(self):
22         return self.user.username
```

Давайте детально розберемо вищенаведену модель:

- бий рядок: декларуємо клас StProfile; це звичайний клас моделі і тому він також унаслідується від базового класу Model; даний клас міститиме додаткову інформацію для користувача і буде прив'язаний до конкретного об'єкта User в базі;
- 9ий: завдяки полю user ми зв'язуватимемо дві моделі User і StProfile; таким чином, маючи об'єкт User ми завжди зможемо отримати його об'єкт профіля через user.stprofile; двосторонній зв'язок ми з вами розглядали в главі про Django моделі, тому надіюсь даний доступ до моделі StProfile для вас на даний момент зрозумілий;

- 11ий: додаємо вкладений клас Meta, щоб надати необхідного вигляду на графічному інтерфейсі в адмін частині;
- 15ий: поле mobile_phone, через яке ми і почали усю нашу затію; тут ми зберігатимемо додаткову інформацію про користувача: номер його телефону;
- 21ий: метод для представлення об'єкта моделі StProfile в адміністративній частині Django; ми просто виводимо користувацьке ім'я, до якого прив'язаний даний профіль.

Як бачите, код доволі простий і його можете зрозуміти на всі 100%, якщо сумлінно освоїли главу про Django моделі і бази даних.

Таким чином, у нас вже є місце, куди ми можемо зберігати додаткові дані про користувачів. Проте, якщо зараз зареєструвати даний клас StProfile для Django адмінки, то отримаємо дві різні моделі (а відповідно і форми редагування) для полів, що знаходяться в моделі User і полів, що знаходяться в моделі StProfile. Погодьтесь, що це не надто зручно додавати номер телефону користувача на зовсім іншій формі в адмінці!

В ідеалі усі поля користувача повинні редагуватись на єдиній формі. І у цьому нам знову ж таки допоможе Django фреймворк роботи з моделями. А саме його розширені можливості з кастомізації форм в адміністративній частині.

Django адмінка дозволяє так організувати адмін клас редагування моделі, щоб на формі редагування показувати також поля інших моделей, що мають зв'язок із поточною моделлю. Для цього фреймворк дає клас `InlineModelAdmin`³⁹⁰, який реалізований у двох форматах: `TabularInline` та `StackedInline`. Обидва відрізняються лише форматом виводу даних на формі. Ми скористаємося саме `StackedInline` варіантом.

Створіть ще один порожній файл в корені проекту. Назвемо його `admin.py`, оскільки він міститиме кастомний клас адміністрації User моделі. Завдяки імені модуля `admin.py` Django фреймворк автоматично підчепить усі зміни. Це ви уже також знаєте.

Ось вміст даного файлу:

³⁹⁰<http://djbook.ru/rel1.7/ref/contrib/admin/index.html#django.contrib.admin.InlineModelAdmin>

Заставляємо відобразитись нове поле профіля зручним чином в адмінці

```
1 from django.contrib import admin
2 from django.contrib.auth import admin as auth_admin
3 from django.contrib.auth.models import User
4
5 from .models import StProfile
6
7
8 class StProfileInline(admin.StackedInline):
9     model = StProfile
10
11 class UserAdmin(auth_admin.UserAdmin):
12     inlines = (StProfileInline,)
13
14 # replace existing User admin form
15 admin.site.unregister(User)
16 admin.site.register(User, UserAdmin)
```

Важливо детально оглянути усі рядочки, оскільки тут є багато нового для нас:

- 8ий рядок: даний клас зв'яже модель StProfile із формою редагування моделі User в адмінці; він унаслідується від StackedInline класу; StackedInline дозволить показати елементи інших моделей одразу після основних полів моделі User;
- 9ий: завдяки атрибуту `model` вказуємо на модель, яку хочемо підв'язати до форми редагування моделі User;
- 11ий: а це уже клас, який відповідатиме за форму редагування об'єкта User в адмінці; по-суті, ним ми хочемо перекрити уже існуючу форму User в адмінці; тому унаслідуємось від UserAdmin форми, що міститься в `django.contrib.auth` аплікації і далі всередині класу дописуємо те, що хочемо змінити;
- 12ий: адмін форми можуть мати атрибут `inlines`³⁹¹; даний атрибут є списком класів, що зв'язують інші моделі із формою редагування основної моделі; саме тут ми вказали в списку наш клас StProfileInline; це

³⁹¹<http://djbook.ru/rel1.7/ref/contrib/admin/index.html#django.contrib.admin.ModelAdmin.inlines>

заставить форму редагування User показувати поля моделі StProfile на-прикінці сторінки;

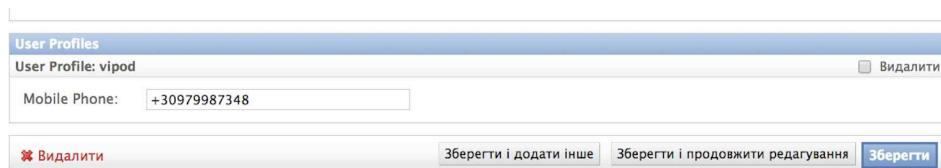
- 15ий: оскільки адмін форма моделі User уже існує в системі, ми маємо спочатку її деактивувати, а потім зареєструвати нашу;
- 16ий: саме тут ми реєструємо нашу кастомну форму редагування моделі User.

Ми створили клас моделі StProfile і прив'язали кастомну форму до моделі User. Дані зміни потрібно синхронізувати у базу даних:

Запускаємо міграцію

- 1 \$ cd /data/work/virtualenvs/studentsdb/src/studentsdb
- 2 \$ python manage.py migrate

Запустіть ваш розробницький Django сервер і відвідайте адміністративну частину Django. Серед списку доступних моделей не повинно бути моделі StProfile, адже ми не реєстрували для неї адміністративної форми. Натомість відвідайте форму редагування будь-якого із існуючих користувачів. Приблизно таку картинку ви повинні знайти наприкінці сторінки:



Нове поле на формі редагуванні користувача в адмініці

Тепер ви можете заповнити номер телефону даного користувача і зберегти зміни. Вони нам пригодяться у наступному кроці.

Оновлюємо сторінку профіля

В даній секції книги нам залишилось останнє завдання: додати нове поле користувача до сторінки профіля.

Для цього знову повертаємося до шаблону profile.html і оновлюємо нашу таблицю із даними. Наводжу лише ті частини шаблону, де відбулися зміни:

Додаємо телефон до профіля користувача

```
1 ...
2 {% block content %}
3
4 <table class="table table-striped">
5 ...
6 ...
7
8     {% if user.stprofile.mobile_phone %}
9         <tr>
10             <td>{{ trans "Mobile Phone" }}:</td>
11             <td>{{ user.stprofile.mobile_phone }}</td>
12         </tr>
13     {% endif %}
14 ...
15 ...
16
17 </table>
18
19 {% endblock content %}
```

Зверніть увагу як ми дістаємо профіль користувача з об'єкта user через атрибут stprofile. Це можливо завдяки полю одностороннього зв'язку OneToOneField в класі моделі StProfile.

Всередині тегу таблиці ми додали ще один рядок, що містить мобільний телефон користувача.

Зайдіть на сайт під користувачем, якому ви додали телефон на формі редагування. Перейдіть на сторінку профіля і переконайтесь, що телефон з'явився всередині таблиці.

На домашнє завдання додайте ще кілька полів до профіля користувача:

адреса, номер паспорта і зображення. Увага: робота із зображенням буде дещо складніша. Але я впевнений, що вам вдасться.

Реалізуйте форму редагування профілю користувача, яка буде доступна самому користувачеві. Таким чином, не лише адміністратор зможе редагувати деталі по окремих користувачах. Для цього можете додати дві закладки на сторінці профіля користувача: Профіль та Редагувати. Закладка Профіль показуватиме поточний профіль, а закладка Редагувати переведитиме користувача на форму редагування його деталей.

Ще одне інтеграційне завдання. До нас вже давно придумали масу Django аплікацій, які дозволяють мати додаткові дані по користувачах в системі. Однією із популярних аплікацій є django-profiles. Поекспериментуйте і спробуйте інтегрувати її форми і моделі по роботі із додатковими даними користувачів. Попереджаю: перед будь-якими експериментами робіть бекап вашої бази даних. Зміни будуть великі і часто безповоротні.

На цьому ми закінчуємо із профілем користувача та його додатковими полями. Далі на нас чекає ще одна не менш цікава секція: інтеграція автентифікації через зовнішні сервіси.

Інтегруємо Facebook логування

В наш час середньостатистичний користувач інтернету має власні зареєстровані користувачі щонайменше на 10-х веб-сайтах, якими він часто користується. А більш технічні користувачі, особливо такі як ми з вами, можуть мати і до 50 часто використовуваних веб-сервісів.

Запам'ятовувати деталі входу на кожен сайт є доволі складно. Мати єдиний пароль для доступу на усі веб-сайти є небезпечно. Вводити усі паролі у додаткову програму для збереження паролей є одним із найпопулярніших рішень. Особливо, якщо вона інтегрована із вашим веб-переглядачем і може автоматично заповнити для вас форми логування на ваших повсякденних онлайн сервісах.

Щоб зекономити ваш час і спонукати до заведення власного користувача на черговому веб-сайті, вже давно придумали метод входу і реєстрації без введення власних даних. Натомість ці дані беруть з одного із веб-сайтів, де ви уже маєте зареєстрованого користувача. Зазвичай підтримуються лише найбільш популярні онлайн ресурси і портали такі як Google, Facebook, Twitter і тому подібні.

При першій спробі залогуватись на новий веб-сайт з використанням ваших даних із іншого сервісу, вам автоматично створять новий екаунт на новому сайті і залогують. Якщо ж ви уже маєте користувача, тоді просто відбудеться логін.

Бувають навіть такі інтеграції логінів через зовнішні сервіси, які не запам'ятовують ваших даних в базі даних (тобто не створюють користувача в базі), а лише тимчасово створюють вашу користувацьку сесію, яка з часом (або з виходом із сайту) зникає.

В даній секції ми також інтегруємо логін через зовнішній сервіс. Ним буде соціальна мережа Facebook, яка надає ряд протоколів для даної інтеграції. На користувацький інтерфейс ми з вами виведемо лінк “via Facebook” (“через Facebook”), який:

- сконтактує ваш Facebook екаунт;
- попросить вашого дозволу на отримання даних з вашого існуючого Facebook екаунту;
- створить нового користувача в базі даних нашої applікації з деталями отриманими із Facebook мережі;
- і одразу залогує вас на веб-сайт під новоствореним користувачем.

При усіх наступних використаннях лінка “via Facebook” вас логуватимуть під існуючим користувачем.

Як уже зазначалось раніше, для даної інтеграції ми не писатимемо власного Python коду, що напряму контактуватиме Facebook мережу. Натомість скористаємося існуючою аплікацією `python-social-auth`. Вона надає інтеграцію логіну з допомогою більше, ніж 70 онлайн сервісів. Також даний пакет містить інтеграцію для інших Python веб-фреймворків: Pyramid, Flask, Tornado, Webpy.

Інсталюємо `python-social-auth`

Почнемо з інсталяції пакету [python-social-auth³⁹²](#) та інтеграції його у якості Django аплікації.

Відкриваємо файл із списком додаткових пакетів і дописуємо туди `python-social-auth`:

`requirements.txt`

```
1 Django==1.7.2
2 MySQL-python==1.2.5
3 Pillow==2.6.1
4 django-crispy-forms==1.4.0
5 python-dateutil==2.3
6 django-registration-redux==1.2
7 python-social-auth==0.2.12
```

Інсталюємо новий пакет:

Інсталюємо `python-social-auth`

```
1 $ pip install -r requirements.txt
```

Перед використанням заіnstальованого пакету маємо пройтись по налаштуваннях і додати необхідні опції. [Документація пакета³⁹³](#) детально описує необхідні кроки:

³⁹²<http://python-social-auth.readthedocs.org/en/latest/>

³⁹³<http://python-social-auth.readthedocs.org/en/latest/>

Налаштовуємо аплікацію python-social-auth, settings.py

```
1 INSTALLED_APPS = (
2     ...
3     'registration',
4     'social.apps.django_app.default',
5     'students',
6     'studentsdb',
7 )
8
9 TEMPLATE_CONTEXT_PROCESSORS = \
10     global_settings.TEMPLATE_CONTEXT_PROCESSORS + (
11     "django.core.context_processors.request",
12     "social.apps.django_app.context_processors.backends",
13     "social.apps.django_app.context_processors.login_redirect",
14     "studentsdb.context_processors.students_proc",
15     "students.context_processors.groups_processor",
16 )
17
18 AUTHENTICATION_BACKENDS = (
19     'social.backends.facebook.FacebookOAuth2',
20     'django.contrib.auth.backends.ModelBackend',
21 )
22
23 SOCIAL_AUTH_FACEBOOK_KEY = ''
24 SOCIAL_AUTH_FACEBOOK_SECRET = ''
```

Ось основні налаштування пакету python-social-auth:

- 4ий рядок: додаємо Django аплікацію до списку заінстальованих у нашому проекті; аплікація знаходиться під адресою social.apps.django_app.default; окрім default варіанту пакет також надає аплікацію, яка зберігатиме дані в NoSQL базу MongoDB; але нам потрібен звичайний Django ORM, тому обираємо варіант default;

- 12ий: для коректної роботи social аплікації потрібно додати її два процесори контексту; backends процесор робить доступним список автентифікаційних бекендів в шаблонах;
- 18ий: дана змінна представляє список автентифікаційних бекендів; кожен із бекендів вміє “впускати” користувачів на веб-сайт, які існують в різних місцях; це може бути локальна Django база, LDAP сервер, соціальна мережа, файл у файловій системі, користувачі одного із віддалених серверів і т.д.; таким чином, Django дозволяє підключати кілька зовсім різних джерел користувачів одночасно;
- 19ий: FacebookOAuth2 бекенд відповідає за логування Facebook користувачів використовуючи його протокол OAuth2;
- 20ий: дефолтний бекенд ModelBackend ми також підключили, щоб не втрачати можливість “впускати” Django користувачів на сайт, а також в адмінку сайту;
- 23ий: ці дві змінні вказують на публічний і приватний ключі Facebook аплікації, через яку відбудутимуться входи на наш сайт; їх ми налаштуємо пізніше.

Тепер нам потрібно запустити міграцію, щоб нова аплікація створила усі необхідні моделі і дані в базі:

Синхронізуємо останні зміни проекту в базу

```
1 # як завжди, не забудьте активувати віртуальне середовище
2 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb
3 $ python manage.py makemigrations
4 $ python manage.py migrate
```

Після цього перевірте адміністративну частину Django. Мала б з’явитись нова група моделей під назвою Python Social Auth. Якщо є, тоді рухаємось далі. Інакше перевірте усі попередні кроки і переконайтесь, що команди інсталяції та міграції завершилися успішним виводом без помилок.

Додаємо посилання Facebook логіну

Наступним кроком з інтеграції social аплікації буде підключення її URL шаблонів³⁹⁴:

Підключаємо URL шаблони із python-social-auth, urls.py

```

1 urlpatterns = patterns('',
2     ...
3     # Social Auth Related urls
4     url('^social/', include('social.apps.django_app.urls', namespace\
5     ='social')),
6     ...
7 )

```

З даних шаблонів нас цікавить лише один під назвою begin. Він, власне, і контактує зовнішній сервіс, отримує від нього дані користувача, створює користувача в Django базі і потім його логує. Все в одній в'юшці.

Відкриваємо наш старий добрий шаблон base.html і додаємо лінк логування через Facebook до анонімної панелі користувача:

Додаємо лінк Facebook логіну, base.html

```

1 ...
2     <!-- User's toolbar for anonymous users -->
3     <div class="col-xs-6" id="user-toolbar">
4         <a href="{% url 'social:begin' 'facebook' %}?next={{ request\
5 .path }}>{% trans "via Facebook" %}</a> |
6         <a href="{% url "users:auth_login" %}">{% trans "Login" %}</\\
7 a> |
8         <a href="{% url "users:registration_register" %}">{% trans "\\
9 Register" %}</a>
10    </div>
11 ...

```

³⁹⁴<http://python-social-auth.readthedocs.org/en/latest/configuration/django.html#urls-entries>

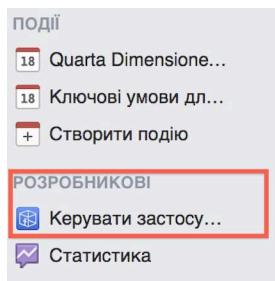
Як бачите, ми додали ще один лінк до користувацької панелі аноніма: “via Facebook”. В якості посилання скористались URL шаблоном “social:begin”. Він приймає назву онлайн сервісу, через який здійснюватиметься логін. У нашому випадку це ‘facebook’. Також додатковим параметром в URL адресі ми передаємо next, якому присвоюємо шлях поточної сторінки. Це потрібно для того, щоб після успішного логування повернути користувача на ту сторінку, де він був перед натисканням лінка ‘via Facebook’. Це значно полегшить життя користувача.

Реєструємо аплікацію у Facebook

Здається ніби усе готово, але насправді ще бракує одного кроку. Більшість онлайн сервісів можуть логувати користувачів на вашому сайті лише після додаткових налаштувань на своїй стороні.

У випадку із сервісом Facebook нам потрібно зареєструвати нову Facebook аплікацію, налаштувати її під адресу свого сайту і скопіювати необхідні ключі даної аплікації в модуль налаштувань нашого проекту.

Отже, заходимо у сервіс Facebook, логуємось і крутимо вниз сторінки допоки в бічній панелі не знаходимо меню “Розробників” з посиланням “Керувати застосунками”:



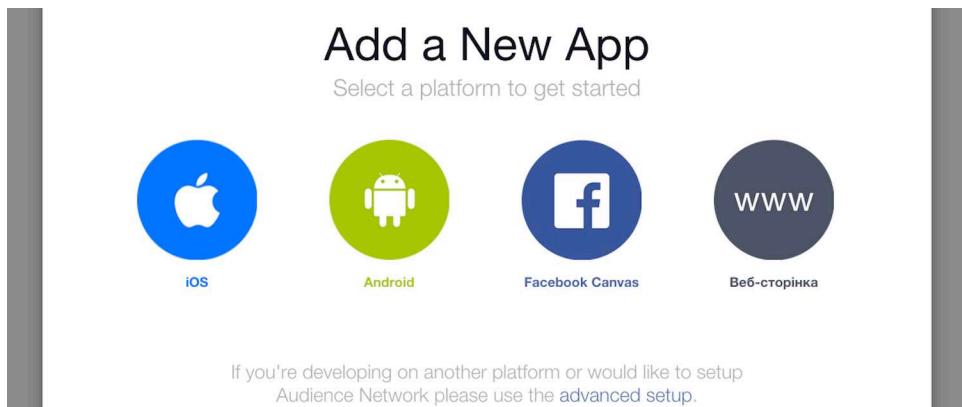
Заходимо в Facebook розробницькі аплікації

На новій сторінці з меню “My Apps” обираємо елемент “Add a New App”, щоб почати процес створення нової аплікації:



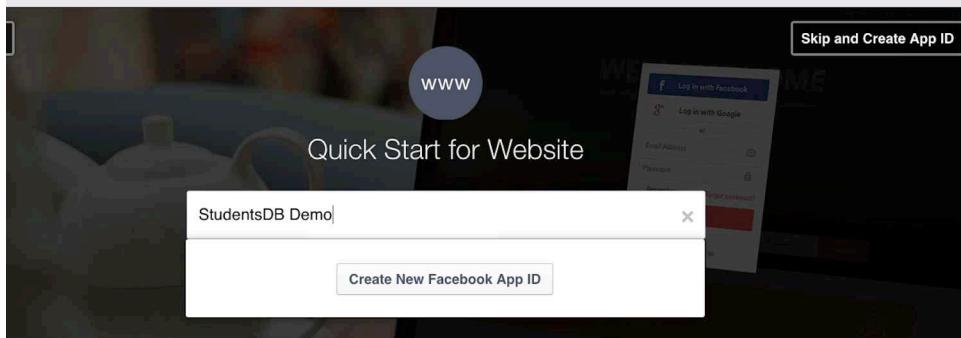
My Apps меню

У вікні, що з'явилось вибираємо “Веб-сторінка”:



Обираємо тип аплікації

На наступному кроці вводимо ім'я для нашої аплікації. Назовемо її “StudentsDB Demo”:



Даємо ім'я аплікації

Тепер маємо обрати категорію для аплікації, а також визначити чи це буде тестова аплікація. Для наших експериментів нам знадобиться тестова аплікація. Але кожна тестова аплікація повинна мати продакшин варіант. Тому на даному етапі ми створюємо не тестову аплікацію. Пізніше нам прийдеться пройти ще раз через процес створення нової Facebook аплікації, щоб створити тестову аплікацію для поточної аплікації.

Отже, заповнюємо форму і натискаємо кнопку “Create App ID”:

Create a New App ID

Create **StudentsDB Demo** App?

Hi Is this a test version of another app? [Дізнатися більше.](#)

Категорія

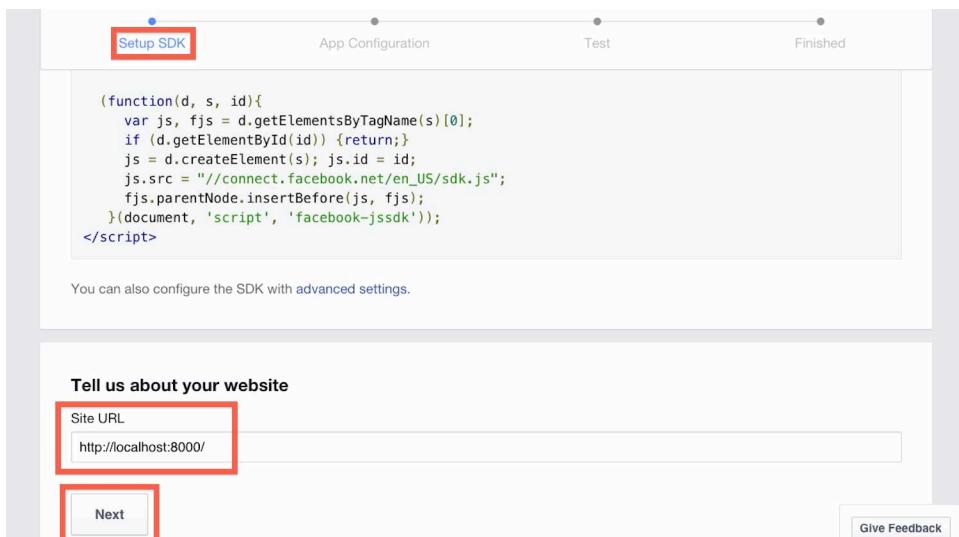
Навчання

By proceeding, you agree to the Facebook Platform Policies

Скасувати Create App ID

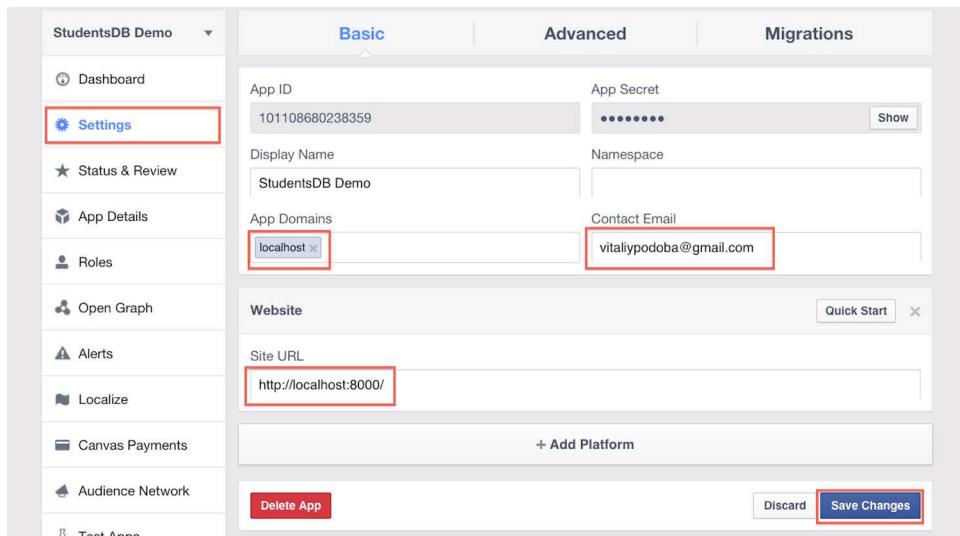
Створюємо аплікацію

В наступному кроці сервіс пропонує нам скопіювати кусок Javascript коду для фронт-енд інтеграції логіну. Але нам поки даний варіант не потрібен, тому ми скролимо у кінець сторінки і просто натискаємо кнопочку “Next”:



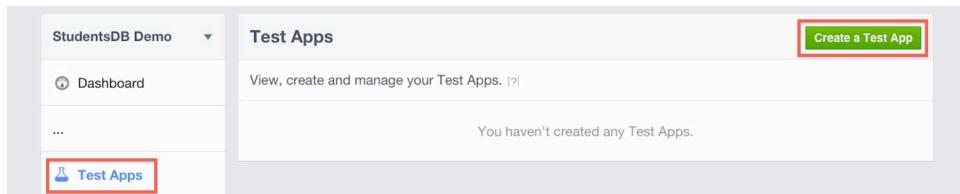
Крок налаштування Javascript SDK

Тепер аплікація є створена. Щоб потрапити на форму управління даною аплікацією знову скористайтесь меню “My Apps” і знайдіть серед елементів новостворену аплікацію “StudentsDB Demo”. Заходимо на меню налаштувань і переконуємось, що усі налаштування співпадають (звісно емейл заповніть власний). Зверніть увагу на нове поле “App Domains”. Даний список доменів Facebook аплікація підтримуватиме під час логування. Нам поки потрібен лише localhost:



Налаштовуємо аплікацію StudentsDB Demo

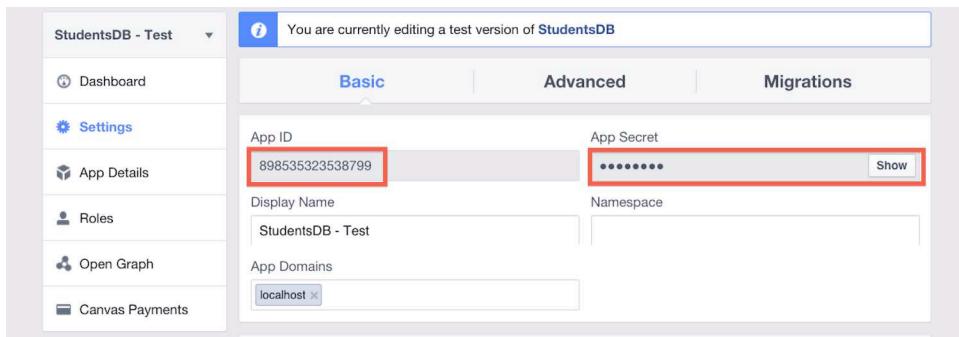
Основна аплікація повністю готова. Але ми не можемо нею користуватись в розробницьких цілях. Вона працює лише для публічних продакшин сайті. Для локальних тестів та розробки нам потрібно також додати тестову аплікацію прив'язану до даної основної. Для цього скористаємось пунктом меню в лівій колонці під назвою “Test Apps”:



Створюємо тестову аплікацію

Аплікацію називаємо “StudentsDB Demo - Test”. Переконайтесь, що усі налаштування тестової аплікації співпадають із налаштуваннями основної: Site URL і App Domains.

Тепер знову переходите на закладку Settings same тестової аплікації і копіюєте необхідні ключі аплікації у settings.py налаштування нашого проекту:



Ключі тестової аплікації

Значення поля “App ID” потрібно скопіювати у змінну SOCIAL_AUTH_FACEBOOK_KEY. Значення поля App Secret копіюємо в змінну SOCIAL_AUTH_FACEBOOK_SECRET.

Ключі Facebook аплікації

-
- 1 SOCIAL_AUTH_FACEBOOK_KEY = '*****'
 - 2 SOCIAL_AUTH_FACEBOOK_SECRET = '*****'
-

Отже, маємо повністю інтегровану аплікацію social із використаним Facebook логіном. Посилання для логування у Facebook додали і тестову аплікацію на зовнішньому сервісі налаштували. Ключі аплікації налаштували в модулі settings.py. Ніби все готово!

Перевантажуємо сервер, сторінку в браузері і тестиємо.

Клацніть по лінку ‘via Facebook’. Повинні отримати подібне вікно:



StudentsDB - Test отримає наступну інформацію: ваші public profile. [?](#)

[Переглянути інформацію, яку Ви надаєте](#)

Ваша згода не дозволятиме застосункові самовільно дописувати у Facebook.

Вікно авторизації права на доступ до ваших даних

Підтверджуєте доступ до ваших даних нашою Django аплікацією і при успішному вході ви повинні автоматично потрапити назад на корінь нашого веб-сайту. Але тепер у залогованому стані. Тепер зайдіть на сторінку свого профіля і перегляньте деталі новоствореного користувача. Маєте побачити, що користувацьке ім'я було автоматично згенероване для вас і має формат <Ім'я><Прізвище><хеш код>. У моєму випадку я отримав username рівний стрічці: "VitaliyPodoba5b54766492a24213".

Таким чином, python-social-auth створив для нас користувача в Django базі наповнивши його даними із нашого Facebook профілю.

Якщо у вас не працює логування у Facebook і щось ламається при контакті зовнішнього сервісу, тоді рекомендую переглянути налаштування тестової аплікації у Facebook.

Якщо зовсім все складно із дебагом Facebook логіну, тоді звертайтесь у закриту групу підтримку, що йде разом із Рекомендованим пакетом книги. Там вам допоможуть розв'язати проблеми і нададуть кілька корисних підказок.

На домашку пропоную вам перетворити текстовий лінк ‘via Facebook’ на графічну кнопочку. Подібні кнопки-лінки можете бачити на інших веб- сайтах, що пропонують Facebook логін.

Також додайте ще два лінки до користувацької панелі аноніма: ‘via Twitter’ і ‘via Google+’. Підключіть обидва логіни через дані сервіси. Для цього вам також прийдеться пройти через процес налаштування додаткових аплікацій на цих соцмережах. Ну і звісно, також прийдеться переглянути усі налаштування нашого проекту і скопіювати необхідні ключі.

І не забувайте про регулярні коміти в репозиторій коду.

Домашнє завдання

Тема користувачів та авторизації є доволі обширною. Проте, як ви вже зрозуміли, у світі Django існує велика кількість якісних інструментів. В більшості випадків на реальних проектах вам їх буде достатньо. І винаходити велосипед не прийдеться.

Тому вам залишаються дві речі:

- розуміти основні принципи роботи із користувачами: автентифікація, авторизація, методи логування і т.д.;
- знати про існуючі Django рішення і вміти інтегрувати їх у свої проекти.

...

На домашнє завдання, окрім задач, що були поставлені протягом глави, пропоную спробувати справитись також із наступними завданнями. Починаємо від простіших, закінчуємо важчими:

- локалізуйте нові елементи графічного інтерфейсу на українську мову;
- реалізуйте ще одну закладку в нашій аплікації: Користувачі; вона показуватиме список існуючих користувачів в базі; також кожен автентифікований користувач може заходити на профіль іншого користувача з допомогою даного списку;
- на власній формі логування додайте лінк “Забули пароль?” і самостійно організуйте процедуру оновлення паролю через емейл;
- реалізуйте користувачам можливість самим змінювати пароль будучи залогованими на сайті;
- “закопайтесь” з головою в пакет python-social-auth; присвятіть кілька годин навігації по коду; ви здивуєтесь тим рівнем абстракції, на якому реалізований даний пакет; знайте: розуміння подібного коду - це вже новий рівень для програміста; починайте лише з того коду, що має відношення до Django фреймворку; підчитуйте документацію і неодноразово повертайтесь до розбору коду даного пакету в якості вправи; з кожним наступним разом ви більше розумітимете, що там відбувається;
- реалізуйте Facebook логін самостійно через їхню Javascript SDK; на даний момент ми маємо варіант серверного логіну;
- спробуйте розібратись із протоколом OAuth2 (через нього працює Facebook логін) і самостійно, без використання python-social-auth, реалізувати код, що робитиме повний логін через Facebook; якщо буде важко (а я

думаю, що буде), перегляньте код аплікації `django-facebook-connect`³⁹⁵ та почерпніть кілька ідей.

Якщо розв'яжете усі вищеперелічені завдання, тоді зможете не лише з легкістю використовувати існуючі інтеграції користувачів, але й винаходити власний велосипед ;-)

...

В наступній главі ми з вами зачепимо кілька більш розширених аспектів Django фреймворків. Розберемо, що таке мідлвара, Django скрипт. Навчимось писати власний шаблонний тег, а також кастомний фільтр.

³⁹⁵ <https://github.com/noamsu/django-facebook-connect>

13. Додатковий функціонал: мідвара, команда, кастомний тег та фільтр

На даний момент ми з вами оглянули усі основні аспекти Django фреймворка. Загалом Django надає масу різноманітних мікро-фреймворків та можливостей для кастомізації під ваші потреби. Але, щоб почати створювати веб-сайти з його допомогою та претендувати на позицію джуниора Python веб-розробника вам і не потрібно знати все. Згодом, на реальних проектах, ви доволі швидко заповнюватимете прогалини у знаннях про Django.

Тим не менше, в даній главі ми оглянемо ще кілька речей, які варто знати про фреймворк навіть початківцю:

- поняття мідвари (англ. middleware);
- команди django-admin скрипта;
- реалізація власних тегів та фільтрів для шаблонів.

Дані знання ми застосуємо у не надто складних практичних завданнях, де:

- реалізуємо власну мідвару, яка обчислюватиме час генерації відповіді на сервері і відображатиме на сторінці;
- напишемо команду, що відображатиме статистичні дані про нашу базу;
- створимо кастомний шаблонний фільтр для приведення стрічок в цілі числа;
- закодимо простий кастомний шаблонний тег, що вставлятиме елемент посторінкової навігації на сторінці.

Кожна із наступних під-секцій міститиме теоретичний вступ про нове поняття, за яким слідуватиме практичне завдання. І як завжди, я не забув про домашні завдання.

А почнемо ми із розбору поняття мідвари:

Django мілдвара

Мілдварами ми з вами користуємось із самого початку проекту, починаючи з глави про динамізацію головної сторінки. Ми навіть кілька разів згадували їх, коли працювали із користувачами в Django фреймворку. І от врешті-решт, розберемо дане поняття детально.

Система мілдвар (з англ. middleware - проміжний, по середині) в Django - це міні-фреймворк плагінів, які безпосередньо і глобально впливають на усі записи та відповіді, що відбуваються на веб-сайті. А сама мілдвара - це компонента, що є відповідальною за певну специфічну функцію, що запускається під час обробки запиту чи формування відповіді із сервера.

Наприклад, мілдвара [LocaleMiddleware³⁹⁶](#) додає необхідні мовні заголовки до відповіді. Цими заголовками вона повідомляє про мову контенту, що йде із сервера на клієнт. А мілдвара [AuthenticationMiddleware³⁹⁷](#) створює об'єкт користувача і присвоює його в якості атрибута на кожен запит. Таким чином, ми маємо доступ до `request.user` із наших в'юшок.

На даний момент ми з вами користуємось великою кількістю вбудованих та зовнішніх (що прийшли із додаткових аплікацій) Django мілдвар. Дані компоненти активуються з допомогою змінної `MIDDLEWARE_CLASSES` в налаштуваннях проекту. Вона містить список стрічок, кожна з яких є шляхом до класу мілдвари:

Наш поточний список мілдвар в проекті

```
1 MIDDLEWARE_CLASSES = (
2     'django.contrib.sessions.middleware.SessionMiddleware',
3     'django.middleware.locale.LocaleMiddleware',
4     'django.middleware.common.CommonMiddleware',
5     'django.middleware.csrf.CsrfViewMiddleware',
6     'django.contrib.auth.middleware.AuthenticationMiddleware',
7     'django.contrib.auth.middleware.SessionAuthenticationMiddleware',
8     'django.contrib.messages.middleware.MessageMiddleware',
```

³⁹⁶<http://djbook.ru/rel1.7/ref/middleware.html#module-django.middleware.locale>

³⁹⁷<http://djbook.ru/rel1.7/ref/middleware.html#django.contrib.auth.middleware.AuthenticationMiddleware>

```
9     'django.middleware.clickjacking.XFrameOptionsMiddleware',
10 )
```

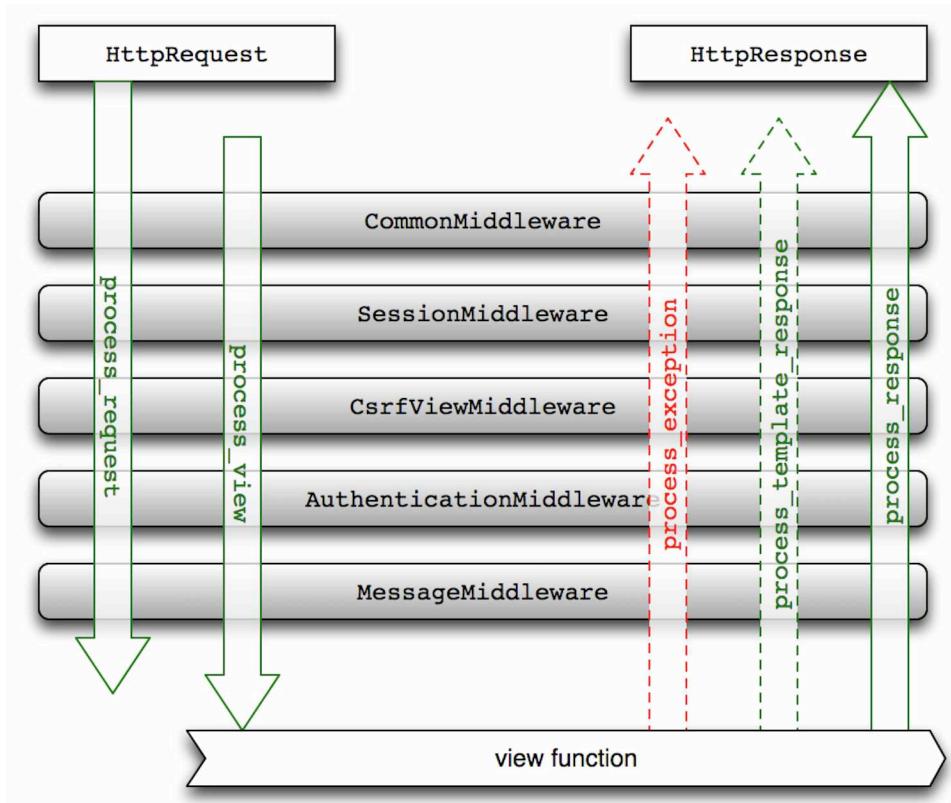
Частина з цих компонент забезпечує базовий рівень безпеки і захист від певного роду атак. Інша частина допомагає реалізувати систему користувачів на сайті. Решта: переклади та локалізацію.

Порядок даних мідлвар є важливим. Список MIDDLEWARE_CLASSES працюється на кожних запиті та відповіді. При запиті дані компоненти запускаються по черзі згори до низу. Після формування відповіді вони запускаються у зворотньому порядку.

Також компоненти можуть залежати одна від одної. Наприклад Authentication Middleware обов'язково має бути в списку не раніше, ніж SessionMiddleware, адже перша використовує інформацію сесії, щоб отримати і встановити об'єкт залогованого користувача.

Анатомія мідлвари

Перед тим, як братись за написання власної компоненти, потрібно розібратись із її структурою та можливостями.



Структура викликів методів мілдвари. Зображення взяте із офіційного сайту djngoproject.com

Мілдвара є звичайним Python класом з набором (або лише одним із них) визначених методів, що запускаються під час обробки запиту чи формування відповіді.

Під час обробки запиту (request) запускаються методи: `process_request`³⁹⁸ та `process_view`³⁹⁹. Під час обробки відповіді (response) запускаються методи: `process_exception`⁴⁰⁰, `process_template_response`⁴⁰¹ та `process_response`⁴⁰².

Давайте детальніше оглянемо кожен із даних методів.

³⁹⁸ http://djbook.ru/rel1.7/topics/http/middleware.html#process_request

³⁹⁹ https://docs.djangoproject.com/en/1.7/topics/http/middleware/#process_view

⁴⁰⁰ http://djbook.ru/rel1.7/topics/http/middleware.html#process_exception

⁴⁰¹ http://djbook.ru/rel1.7/topics/http/middleware.html#process_template_response

⁴⁰² http://djbook.ru/rel1.7/topics/http/middleware.html#process_response

process_request

process_request отримує єдиний аргумент request, який є об'єктом класу HttpRequest. Даний метод викликається на кожному запиті перед тим як фреймворк вирішить, яку в'юшку запустити.

Метод process_request має повернути None або інстанс класу HttpResponseRedirect. Якщо він повертає None, тоді Django продовжить запускати process_request методи інших по списку мідлвар, далі запустить process_view методи і, наприкінці, саму в'юшку. Якщо ж наш process_request метод поверне HttpResponseRedirect об'єкт, тоді жодні мідлвари, що працюють над обробкою запиту, не будуть запускатись. Навіть сама в'юшка буде проігнорована. І в цьому випадку візьмуться до уваги лише мідлвари обробки відповіді.

Таким чином, з допомогою метода process_request ми можемо повністю змінити зміст відповіді від сервера до клієнта.

process_view

Даний метод приймає наступні аргументи:

- request: об'єкт запиту;
- view_func: функція в'юшки, яку Django запустить після виконання методу process_view;
- view_args: список позиційних аргументів для в'юшки;
- view_kwargs: словник ключових аргументів в'юшки.

process_view викликається безпосередньо перед викликом основної в'юшки запиту. Так само як і метод process_request, метод process_view повинен повернати або None, або об'єкт класу HttpResponseRedirect. Аналогічно, при поверненні None Django продовжує запуск наступних в списку мідлвар по обробці в'юшок та помилок (process_exception). А при значенні HttpResponseRedirect наступні мідлвари обробки запиту, так само як і сама в'юшка, ігноруються. Запускаються лише мідлвари обробки відповіді із сервера. Таким чином, є можливість повністю впливати на запуск основної в'юшки.

process_template_response

process_template_response приймає два аргументи:

- `request`: об'єкт запиту;
- `response`: інстанс класу `TemplateResponse`, який повертає мілдвара обробки запиту або в'юшка.

Даний метод викликається одразу після завершення коду в'юшки. При цьому `response` повинен мати метод `render`. Тобто бути об'єктом класу `TemplateResponse`. Якщо відповідь із в'юшки не є у вигляді шаблону, тоді дана мілдвара проігнорується.

Метод `process_template_response` обов'язково повинен повертати об'єкт `TemplateResponse`. При цьому це може бути той самий `response`, який передали даному методу. В такому випадку можемо просто змінювати `response.template_name` чи `response.context_data` атрибути відповіді. Або можемо повністю підмінити `TemplateResponse` об'єкт на виході.

`process_response`

Даний метод також отримує два аргументи:

- `request`: об'єкт запиту;
- `response`: об'єкт класу `HttpResponse` або `StreamingHttpResponse` згенерований в'юшкою або однією із попередніх мілдвар.

Django фреймворк запускає `process_response` на кожній обробці запиту одразу перед тим, як повернати даний запит браузеру.

Аналогічно до методу `process_template_response`, метод `process_response` також має повернати об'єкт запиту. Але клас запиту має бути або `HttpResponse`, або `StreamingHttpResponse`⁴⁰³. Це може бути той самий `response` переданий методу, або повністю створений новий об'єкт.

Незалежно від того, що повернути мілдвари на етапі обробки запиту, мілдвара `process_response` викликається завжди. Це також означає, що в даному методі `process_response` не можна залежати від дій, що знаходяться в `process_request`.

`process_exception`

І останній метод мілдвари - це `process_exception`. Він також приймає два аргументи:

⁴⁰³<http://djbook.ru/rel1.7/ref/request-response.html#django.http.StreamingHttpResponse>

- `request`: об'єкт запиту;
- `exception`: об'єкт помилки згенерованої в'юшкою.

Цей метод Django викликає при обробці відповіді, якщо в'юшка видала помилку.

`process_exception` має повернати `None` або об'єкт `HttpResponse`. Якщо повертає `None`, тоді виконується стандартна обробка помилок в Django і ви бачите деталі помилки у браузері (якщо це розробницьке середовище). Якщо ж метод повертає інстанс класу `HttpResponse`, тоді Django далі застосовує до нього задекларований список мідлвар з обробки запиту і відправляє клієнту.

...

Ми розібралися із усіма методами, тому тепер давайте реалізуємо частину з них на практиці.

Пишемо власну мідлвару

В даній секції реалізуємо мідлвару, яка обчислюватиме час витрачений на обробку запиту та формування відповіді і виводитиме його на сторінці.

Мідлвари можуть бути де завгодно у коді, адже, щоб їх підключити достатньо згадати шлях до них у списку `MIDDLEWARE_CLASSES` налаштувань проекту. Тим не менше, заведено класти код даних компонент у корінь проекту чи аплікації в модуль під назвою `middleware.py`.

Ви можете підготувати окрему Django аплікацію під назвою `page_stats` (статистика сторінки) і розробляти дану мідлвару саме там. Але ми, щоб не розтягувати дану главу речима, які уже вміємо (а саме створювати і підключати чергову аплікацію у проект), створимо модуль `middleware.py` прямо в корені проекту `studentsdb`:

Створюємо файл middleware.py в корені проекту

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb
2 $ touch middleware.py
```

Для того, щоб порахувати час від початку обробки запиту і до отримання об'єкту відповіді ми скористаємось доволі простим і примітивним підходом. На етапі обробки запиту, а саме в методі `process_request` нашої мілдвари, ми запам'ятаемо поточний час як атрибут на об'єкті запиту. А на етапі обробки відповіді, вже після того як в'юшка відпрацювала і віддала HTML код сторінки, всередині методу `process_response` нашої мілдвари, ми порахуємо час, що минув з моменту виклику методу `process_request` і виведемо його на сторінку.

Так, даний метод не є абсолютно точним, адже не враховує часу витраченого самим Django фреймворком на формування об'єкту запиту (`request`) та виклик інших мілдвар перед відправкою відповіді браузеру. Але цього буде достатньо, щоб продемонструвати концепцію мілдвар в Django.

Отже, відкриваємо новостворений модуль для мілдвари і наповнюємо необхідним кодом:

Мілдвара для обчислення часу на обробку запиту та створення відповіді

```
1 from datetime import datetime
2
3 from django.http import HttpResponseRedirect
4
5
6 class RequestTimeMiddleware(object):
7     """Display request time on a page"""
8
9     def process_request(self, request):
10         request.start_time = datetime.now()
11         return None
12
13     def process_response(self, request, response):
14         # if our process_request was canceled somewhere within
```

```
15      # middleware stack, we can not calculate request time
16      if not hasattr(request, 'start_time'):
17          return response
18
19      # calculate request execution time
20      request.end_time = datetime.now()
21      if 'text/html' in response.get('Content-Type', ''):
22          response.write('<br />Request took: %s' % str(
23              request.end_time - request.start_time))
24
25      return response
26
27  def process_view(self, request, view, args, kwargs):
28      return None
29
30  def process_template_response(self, request, response):
31      return response
32
33  def process_exception(self, request, exception):
34      return HttpResponse('Exception found: %s' % exception)
```

В класі нашої мілдвари є задекларовані всеможливі методи, але нам потрібні лише перші два. Решту я додав без спеціальної логіки. Для того, щоб ви зорієнтувались як повинні виглядати дані методи і які дані повернати на виході. Давайте детальніше розглянемо даний модуль:

- 1ий рядок: стандартна Python бібліотека нам потрібна для обрахунків часу;
- 3ий: HttpResponse клас використаємо для формування відповіді в process_exception методі;
- 6ий: назовемо клас нашої мілдвари RequestTimeMiddleware; даний клас не повинен унаслідуватись від жодного іншого класу; проте в стилі нових Python класів унаслідуємось від `object`⁴⁰⁴;

⁴⁰⁴<http://habrahabr.ru/post/114585/>

- 9ий: даний метод запускається під час обробки запиту і ще до виклику в'юшки, а також до виклику іншого методу мілдвари `process_view`;
- 10ий: в даному методі ми присвоюємо запиту новий атрибут `start_time` із значенням рівним поточному часу;
- 11ий: важливо, щоб метод повертає `None`, а не об'єкт `HttpResponse`, адже нам не треба перекривати результат в'юшки;
- 13ий: після того як відпрацювала в'юшка вступає в дію наш другий метод - `process_response`; в ньому маємо доступ до об'єкта відповіді (`response`), з яким можемо проводити маніпуляції;
- 16ий: як ми уже обговорювали, фреймворк може і не викликати метод `process_request` нашої мілдвари, якщо попередня у списку мілдвара поверне об'єкт `HttpResponse`; в такому випадку нам не вдається обчислити час; саме тому важливо, щоб мілдвара `RequestTimeMiddleware` була першою у списку мілдвар; але навіть тоді ми страхуємо себе перевіркою на атрибут `start_time` в об'єкті запиту;
- 20ий: запам'ятуємо поточний час в атрибут `end_time` об'єкта запиту; ним ми скористаємося пізніше;
- 21ий: на мілдвару попадають абсолютно усі запити, що йдуть на наш сайт; сюди входять і запити по основну сторінку (HTML код домашньої наприклад), і на додаткові ресурси сторінки (Javascript, CSS, файли зображені); наша задача вимірювати час, що затрачається на генерацію основної сторінки, тобто HTML сторінки; саме даний рядок допомагає проігнорувати усе інше; зауважте, що не усі відповіді можуть містити заголовок 'Content-Type', тому дістаємо його із об'єкта `response` через метод `get`; у випадку, якщо тип відповіді не є `html`, або взагалі відсутній, ми ігноруємо дану відповідь;
- 22ий: якщо ж сторінка типу HTML, тоді додаємо до самого її кінця стрічку із часом, що був затрачений від початку запиту до обробки відповіді; просто віднімаємо поточний час від того, що запам'ятали в методі `process_request`; об'єкт `HttpResponse` надає метод `write`, який додає передану стрічку до кінця вмісту сторінки;
- 25ий: обов'язково потрібно повернути об'єкт класу `HttpResponse` з даного методу; в нашему випадку ми повертаємо той самий об'єкт, який отримали в якості аргумента; ми лише його трохи змінили; інакли

приходить повністю створювати новий HttpResponseRedirect і вже повернати його із методу.

- 27ий: даний метод є у нас порожнім і служить чисто демонстраційним цілям; тут можете бачити необхідний мінімум для реалізації даного методу; залишається лише набити його необхідною логікою при потребі;
- 30ий: аналогічним чином наведений метод process_template_response;
- 33ий: і дуже простий варіант обробки помилки, що виникає у основній в'юшці запиту; ловимо помилку і натомість віддаємо об'єкт HttpResponseRedirect із текстом помилки; щоб потестувати даний метод вам придеться у одну із наших в'юшок вставити код, що викидатиме помилку (щось на зразок “raise Exception('error')”) і у браузері зайти на сторінку даної в'юшки; наша мілдвара повинна її перехопити.

Тепер нам потрібно додати нашу мілдвару до списку мілдвар підключених у проекті:

Модуль налаштувань проекту settings.py

```
1 MIDDLEWARE_CLASSES = (
2     'studentsdb.middleware.RequestTimeMiddleware',
3     'django.contrib.sessions.middleware.SessionMiddleware',
4     'django.middleware.locale.LocaleMiddleware',
5     ...
6 )
```

Наш клас ми поставили першим у списку MIDDLEWARE_CLASSES. Він запускатиметься першим при обробці запиту і останнім при обробці відповіді.

Цього достатньо, щоб почати тестувати роботу нашого нового коду. Робіть рестарт Django сервера і оновіть головну сторінку нашої аплікації. Подібна стрічка повинна з'явитись у вас на сторінці:

The screenshot shows a top navigation bar with '3' notifications, a user profile icon, 'Кеню' (Kenya), 'Назар' (Nazir), '212311', and a 'Дія' (Action) dropdown. Below is a page header with a navigation menu ('« 1 2 3 4 5 6 »'). A copyright notice '© 2014 Сервіс Обліку Студентів' and a timestamp 'Request took: 0:00:00.132211' are at the bottom.

Результат роботи нашої мідлвари

Дане завдання вам дало загальне уявлення про те, як мідлвара може впливати на процес обробки запиту та відповіді. А тепер пропоную цілий ряд невеликих завдань, які допоможуть вам задіяти решту методів мідлвари.

Зрозуміло, що виводити час потрачений на запит може мати чисто розробницьку функцію, щоб допомогти девелоперу тримати сторінку в хорошому тонусі та швидкості. На продакшині немає жодного сенсу підключати дану мідлвару. Проте, щоб точно гарантувати, що наша мідлвара не “вилізе” у непотрібний момент, пропоную додати до її коду умову на перевірку DEBUG режиму проекта. Якщо DEBUG включений (тобто ми швидше за все знаходимось в розробницькому середовищі), тоді вставляємо час на сторінку.

На даний момент ми додаємо новий елемент, що містить час, у самий кінець сторінки, після тегу `html`. Це не дуже коректно, адже усі видимі теги сторінки повинні бути в межах тегу `body`. Пропоную переробити метод `process_response` таким чином, щоб вставляти наш елемент із часом всередині `body` тегу. Для цього вам прийдеться розібратись із тим, як в мові Python “парсити” (від англ. parser, синтаксичний аналізатор) HTML код і змінювати його. Підказка: гляньте у вбудований Python парсер `HTMLParser` або додаткову бібліотеку `BeautifulSoup`. Розбір HTML коду з Python коду є дуже поширеним завданням, тому досвід, що набудете в результаті даної домашки точно не буде зайвим!

Коли будете парсити HTML код у мілдварі, вставте не просто стрічку з текстом, а цілий HTML тег. На нього пізніше навішаєте додаткові класи, щоб зробити даний елемент гарнішим. Ось приклад як можна вставити даний елемент на сторінку і зробити його приємнішим для ока:



Приклад кращого форматування елемента часу на сторінці

Ускладніть логіку мілдвари і замість відображення часу на сторінці, у випадку, якщо запит/відповідь пропрацювали більше, ніж одну секунду: виводіть власний текст на сторінку. Даний текст перекриватиме оригінальну сторінку: "Обробка запиту занадто повільна. Розробник - переглянь свій код!". Як це зробити? Ви знаєте. Просто треба повернути власний `HttpResponse` з методу `process_response` або `process_template_response`. Зазвичай весь запит відпрацьовує в межах однієї секунди на локальному інстансі. Тому для тесту кінцевого результату вам можуть пригодитись штучні "галъма": Python модуль `time` із функцією `sleep`. Данна функція зробить паузу у виконанні Python коду на переданий їй час. Також, уся дана затія повинна працювати виключно в розробницькому середовищі, коли "DEBUG = True".

І останнє завдання. Мабуть одне із найважчих. Додайте ще одне мілвару, яка рахуватиме сумарний час витрачений на запити і роботу із базою даних. Також виводіть даний час на сторінці. Тут залишаю вас без підказок. Працюйте!

Якщо “перелопатите” усі ці домашки, повірте, з мілдварами більше проблем не буде. А ми рухаємось далі. І на черзі у нас робота із Django командами.

django-admin команда

Ми з вами уже багато разів використовували [django-admin команди](#)⁴⁰⁵. Це ті команди, які ми запускали через скрипт manage.py: migrate, runserver, startapp, startproject, syncdb, dumpdata і багато інших.

Таким чином, поняття Django команди для нас не є новим. Ми знаємо як їх запускати. А також знаємо те, що команди призначенні для найрізноманітніших цілей: від роботи із базами даних до створення заготовок проектів та запуску тестів чи серверів.

В цій главі ми навчимось створювати власну команду. Данна команда буде доволі простою і дозволить отримувати кількість об'єктів тої чи іншої моделі проекту. Команду назовемо stcount (скорочення від “count students” - порахувати студентів). Ось як зможемо її використовувати:

Приклад використання команди stcount

```
1 $ python manage.py stcount student group user
2 Number of students in database: 16
3 Number of groups in database: 4
4 Number of users in database: 12
```

Ми запустили команду stcount скрипта manage.py і передали йому список назв моделей, що є заіnstальовані у нашому проекті. У відповідь отримали в консоль кількість об'єктів кожної з моделей.

Заготовка команди

Django команда - це Python клас, що унаслідується від [BaseCommand](#)⁴⁰⁶ та повинен мати метод handle і ще кілька атрибутів.

⁴⁰⁵<http://djbook.ru/rel1.7/howto/custom-management-commands.html>

⁴⁰⁶<http://djbook.ru/rel1.7/howto/custom-management-commands.html#django.core.management.BaseCommand>

Крім того, щоб Django автоматично розпізнав команду, вона повинна лежати в заїнсталюваній Django аплікації в підпакеті `management/commands`. Модуль, в якому знаходитиметься клас команди і буде назвою команди.

Команду нашу ми покладемо в аплікацію `students`, адже вона має безпосереднє відношення до моделей даної аплікації.

А почнемо із підготовки пакетів:

Створюємо папки `manegement/commands` з `init` файлами

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students
2 $ mkdir -p management/commands
3
4 # створюємо файл __init__.py в management папці, щоб вона
5 # розпізнавалась Python інтерпретатором як пакет
6 $ cd management
7 $ touch __init__.py
8
9 # те ж саме і з підпапкою commands
10 $ cd commands
11 $ touch __init__.py
12
13 # створюємо модуль для нашої команди stcount.py
14 $ touch stcount.py
```

Нам більше нічого не треба під'єднувати. Django фреймворк автоматично вичитає код з модуля `stcount.py`, адже `students` аплікація є в списку заїнсталюваних в проекті.

stcount в дії

Відкриваємо щойно створений модуль `stcount.py` і додаємо код першої версії команди. Дані версія зможе вивчати кількість студентів в базі:

Перша версія команди stcount

```
1 from django.core.management.base import BaseCommand
2
3 from students.models import Student
4
5
6 class Command(BaseCommand):
7     args = '<model_name model_name ...>'
8     help = "Prints to console number of student related objects in a\
9 database."
10
11     def handle(self, *args, **options):
12         if 'student' in args:
13             self.stdout.write('Number of students in database: %d' %
14                             Student.objects.count())
```

В даній версії ми обробляємо лише модель студента. Тобто виводимо кількість студентів в базі, якщо передали в якості аргумента нашій команді стрічку `student`. Давайте детальніше розглянемо модуль `stcount.py`:

- 1ий рядок: імпортуємо клас `BaseCommand`, від якого унаслідуємось у класі нашої команди;
- 3ий: нам потрібний клас моделі студента, щоб отримати кількість студентів в базі; так, Django команди мають, звісно, доступ до бази даних;
- 6ий: декларація класу команди; назва класу обов'язково має бути `Command` і унаслідуватись від базового класу `BaseCommand`; таким чином, можемо мати в одному модулі не більше, ніж одну команду;
- 7ий: атрибут `args` приймає участь у формуванні документаційної стрічки команди і використовується для виведення інформації про дану команду з допомогою опції “`-help`”; `args` описує список аргументів даної команди; у нашому випадку ми отримуємо одну або більше назв моделей у нижньому реєстрі; наприклад, “`user student group`”;
- 8ий: атрибут `help` надає опис даної команди; також використовується при виведення допоміжної документації про команду;

- 11ий: вся робота команди відбувається в методі handle; він отримує args (спісок аргументів команди, у нашому випадку це спісок моделей, для яких потрібно вивести статистику) та options (спісок опцій переданих команді, наприклад –verbosity, –help і т.д.); наша команда не має жодних власних опцій, і лише унаслідує стандартний набір типу –no-color, –help, –verbosity і тому подібні;
- 12ий: перевіряємо чи стрічка ‘student’ є в спіску переданих аргументів;
- 13ий: якщо так, тоді виводимо кількість студентів в консоль; зауважте при цьому, що ми не використовуємо функцію print, а stdout атрибут об’єкта команди; BaseCommand надає нам два файли виводу: stdout і stderr; варто користуватись саме ними, а не системними файлами вводу та виводу; це допоможе нам пізніше написати тести для команди.

Як бачите, код є доволі простим і доступним. Спробуйте запустити вашу нову команду в консолі і звірте результат:

Тестуємо першу версію команди stcount

```
1 $ python manage.py stcount student
2 Number of students in database: 16
```

Також гляньте на опцію –help вашої нової команди. Побачите, як саме використовуються наші документаційні стрічки задекларовані в класі команди:

Опис команди stcount

```
1 $ python manage.py stcount --help
2 Usage: manage.py stcount [options] <model_name model_name ...>
3
4 Prints to console number of student related objects in a database.
5
6 Options:
7   -v VERBOSITY, --verbosity=VERBOSITY
8           Verbosity level; 0=minimal output, 1=normal \
9           output,
10          2=verbose output, 3=very verbose output
```

```
11    --settings=SETTINGS      The Python path to a settings module, e.g.  
12                                "myproject.settings.main". If this isn't pro\  
13 vided, the  
14                               DJANGO_SETTINGS_MODULE environment variable \  
15 will be  
16                               used.  
17 --pythonpath=PYTHONPATH  
18                               A directory to add to the Python path, e.g.  
19                               "/home/djangoprojects/myproject".  
20 --traceback  
21 --no-color  
22 --version  
23 -h, --help  
                               Raise on exception  
                               Don't colorize the command output.  
                               show program's version number and exit  
                               show this help message and exit
```

Усі опції, які приймає наша команда є унаслідуваними від базового класу `BaseCommand`.

Нагадую, що параметри скрипта, що починаються із однієї або двох рисок, називаються *опціями* команди. Наприклад, так опціями є: `-h`, `--help`, `--version`, `--no-color`. Усі решта параметри називаються *командами* або *аргументами* скрипта.

А тепер спробуйте самостійно подумати та покращити дану команду, щоб вона працювала також із аргументами `'group'` і `'user'`.

Лише після того, як реалізуєте власний варіант, прошу ознайомитись із наступним підходом. У ньому оптимізована обробка аргументів команди з допомогою цикла та нового атрибуту класу `models`:

Покращена версія команди stcount

```
1 from django.core.management.base import BaseCommand
2 from django.contrib.auth.models import User
3
4 from students.models import Student, Group
5
6
7 class Command(BaseCommand):
8     args = '<model_name model_name ...>'
9     help = "Prints to console number of student related objects in a\
10 database."
11
12     models = (('student', Student), ('group', Group), ('user', User))
13
14     def handle(self, *args, **options):
15         for name, model in self.models:
16             if name in args:
17                 self.stdout.write('Number of %ss in database: %d' %
18                                 (name, model.objects.count()))
```

Щоб уникати повторень умовних гілок в коді, у прикладі вище ми скористались циклом. Даний цикл пробігається по заготовленому списку моделей. Таким чином, щоб додати обробку нової моделі в дану команду, достатньо оновити атрибут класу `models`.

На домашнє завдання локалізуйте усі користувачькі стрічки даної команди.

Також пропоную реалізувати ще одну команду під назвою `fill_db`. Данна команда запускатиметься подібним чином: “`python manage.py fill_db -`

`students=4 -groups=5 -users=2`". Даний варіант виклику команди `fill_db` створить в базі 4 нових студента, 5 груп та 2 користувачі. Усі дані новостворених об'єктів будуть рандомно згенеровані. Ця команда допоможе вам з нуля набивати базу в тестових цілях.

На завершення пропоную зайнятись реалізацією доволі крутої і непростої команди. Інколи, коли немає інтернету, а хочеться попрацювати над покращенням нашої веб-аплікації, настає проблема: сторінки не вантажаться коректно, адже використовують зовнішні CSS та Javascript ресурси. В такому випадку приходиться вручну завантажити усі ресурси, що використовуються на сторінках проекту, покласти в потрібну static папочку і оновити посилання на них в необхідних шаблонах. Так от, завдання команди `localize_static` полягає у автоматизації усіх цих дій. Для початку поставте собі за ціль створення команди, яка проаналізує `base.html` шаблон, знайде усі зовнішні CSS та Javascript файли, завантажить їх у static папочки і оновить `base.html` новими static посиланнями. Таким чином, ви у будь-який момент однією командою зможете переходити у офлайн режим. Це буде дуже крута домашка і дасть вам дуже реальний досвід. Лише для тих, хто впертий і не боїться складніших завдань!

...

Одним із популярних застосувань команд є потреба в регулярних скриптах, що виконують певні функції з підтримки продакшин сайтів. Це може бути бекап бази даних, синхронізація даних на зовнішні сервіси і т.д. Більшість із подібних задач уже є реалізовані в самому Django або у додаткових аплікаціях. Проте доволі часто приходиться створювати нові подібні команди під специфічні потреби проекту.

...

На цьому ми завершуємо коротенький огляд django-admin команд у фреймворку Django. І якщо вас цікавлять усі можливості команд та їхні атрибути, звертайтесь на [сторіку документації⁴⁰⁷](#) з розробки кастомних команд.

Кастомні теги

В процесі розробки Django шаблонів для проекту ми освоїли не один шаблонний тег: if, for, url, extend, load, block, with, trans і ще багато інших.

В більшості випадків вбудованих Django тегів вистачає з головою. Проте бувають ситуації, коли логіка в шаблоні ускладнюється і, замість того, щоб постійно перенабирати великий кусок коду в шаблоні, чи переносити багато зайвого коду у в'юшку, розробка власного тегу може вирішити проблему.

Django дозволяє реалізацію власних тегів. І з допомогою тегу load можна завантажити власні теги у шаблони.

В якості прикладу поставимо перед собою задачу перевести шаблон pagination.html на кастомний тег під назвою pagenav. На даний момент ми підключаємо даний елемент через тег include, а змінні, необхідні шаблону pagination.html, він сам собі берез із батьківського шаблону, куди його включають:

Як ми на даний момент підключаємо на сторінку елемент навігації

```
1  {% with object_list=students %}  
2      {% include "students/pagination.html" %}  
3  {% endwith %}
```

А ось як виглядатиме процес додавання навігації на сторінку з наших кастомним тегом:

⁴⁰⁷<http://djbook.ru/rel1.7/howto/custom-management-commands.html>

Використання тегу pagenav

```
1  {% pagenav students is_paginated paginator %}
```

В принципі код не надто спростився. Проте в даному випадку ми явно передаємо усі необхідні змінні тегу pagenav. Крім того, даний тег дозволить нам добре продемонструвати різні варіанти реалізації кастомних тегів в Django фреймворку.

Таким чином, ми спочатку реалізуємо тег повністю своїми силами. А далі скористаємося додатковими Django функціями, які полегшать процес реалізації того ж тегу. А починаємо із підготовки структури аплікації до кастомних тегів.

Заготовка для тегу

Кастомні теги та фільтри повинні знаходитись в застальованій Django аплікації. Аплікація повинна містити підпакет templatetags. Отже, давайте його підготуємо:

```
1  $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students
2  $ mkdir templatetags
3  $ cd templatetags
4  $ touch __init__.py
5  $ touch pagenav.py
```

Дану структуру ми підготували саме в аплікації students, адже поточний шаблон pagination.html уже знаходить там.

Django фреймворк автоматично просканує папку templatetags на спробі першого завантаження тегів в шаблоні. Щоб дана папка додалась до реєстру папок, що містять кастомні теги, потрібно, щоб аплікація була серед заінсталюваних в проекті: INSTALLED_APPS.

Всередині пакету templatetags ми створили модуль pagenav.py. Саме за назвою модуля можна пізніше завантажувати кастомні теги в шаблон:

Ось як можна буде завантажити теги з модуля pagenav.py

```
1  {% load pagenav %}
```

Ми підготували усе для реалізації самого тегу.

Етап компіляції

Django шаблони працюють у два етапи: етап компіляції (compilation) та етап виконання (render).

Етап компіляції аналізує код шаблону, перетворює кожен із шаблонних тегів в об'єкт класу `django.template.Node`. І кожен із даних нодів (з англ. Node - це вузол, елемент) має метод `render`, який запускається на етапі виконання шаблону. Таким чином, після компіляції маємо набір нодів. Під час виконання шаблону викликається метод `render` кожного із нодів і результат складається в одну стрічку.

Даний двоетапний підхід дозволяє зекономити час на повторну компіляцію. Адже аналіз шаблону займає найбільше часу у всьому процесі.

Таким чином, щоб створити кастомний тег, потрібно реалізувати обидва етапи:

- функцію компіляції і підготовки кастомного нода (об'єкта, що унаслідується від `Node`);
- клас самого нода з методом `render`.

Кожного разу, коли шаблонний компілятор зустрічає Django тег, він запускає відповідну функцію компіляції даного тегу і передає їй два аргументи: об'єкт шаблонного парсера (аналізатора, `parser`) та об'єкт стрічки тега (`token`).

Відкриваємо в редакторі модуль `pagenav.py` і починаємо з функції, яка забезпечить етап компіляції нашого тегу:

Функція компіляції тегу pagenav

```
1 from django import template
2
3 register = template.Library()
4
5 # Usage: {% pagenav students is_paginated paginator %}
6
7 @register.tag
8 def pagenav(parser, token):
9     # parse tag arguments
10    try:
11        # split_contents knows how to split quoted strings
12        tag_name, object_list, is_paginated, paginator = token.split\
13        _contents()
14    except ValueError:
15        raise template.TemplateSyntaxError("%r tag requires 3 argume\
16 nts" %
17                                         token.contents.split()[0])
18
19    # create PageNavNode object passing tag arguments
20    return PageNavNode(object_list, is_paginated, paginator)
```

В даному прикладі ми реалізували функцію pagenav, яка розбирає аргументи тегу pagenav передані їй із шаблону в якості аргументу token. Далі вона передає їх в об'єкт класу PageNavNode. Даний клас ми реалізуємо на наступному етапі. А тепер давайте детальніше проаналізуємо код:

- 1ий рядок: пакет template ми дуже часто використовуватимемо в даному модулі; він допоможе нам реєструвати кастомний тег, працювати із шаблонами, змінними та нодами;
- 3ій: створюємо об'єкт бібліотеки тегів та фільтрів Library; присвоюємо його змінній register; далі ми використовуватимемо її, щоб реєструвати кастомні теги;
- 5ий: коментар, що пояснює, як користуватись даним тегом;

- 7ий: декоруємо функцію компілятора з допомогою функції `tag`⁴⁰⁸; ця процедура зареєструє функцію `pagenav` як кастомний тег за одноіменною назвою;
- 8ий: розпочинаємо реалізацію функції `pagenav`; вона отримує об'єкт шаблонного аналізатора та повну стрічку тега;
- 10ий: процедуру розбору аргументів тегу ми огортаємо в оператор `try`, щоб виводити спрощене повідомлення про помилку; це допоможе розробнику шаблонів під час його роботи;
- 12ий: об'єкт `token` має метод `split_contents`, який коректно розбиває стрічку тега на назву тега та усі передані йому аргументи; назва тега йде першою в стрічці, а далі усі аргументи; одразу присвоюємо отримані компоненти по змінних;
- 14ий: якщо кількість аргументів більша або менша, ніж 3, тоді отримаємо помилку; дану помилку підміняємо своєю; тип помилки `TemplateSyntaxError`, адже свідчить про некоректне використання тегу; зауважте, що для отримання назви тегу, в якому знайдено помилку, ми скористалися не `tag_name` змінною, а ще раз розбили текст об'єкта `token` на компоненти і взяли перший із них; при спрацюванні гілки `except` змінна `tag_name` не буде задекларована;
- 20ий: ініціюємо об'єкт класу `PageNavNode` і передаємо йому усі отримані аргументи від тегу `pagenav`; у наступному кроці ми реалізуємо даний Python клас.

Етап рендеру

На даний момент нам бракує класу `PageNavNode`, який відповідає за етап виконання тегу. Даний клас повинен унаслідуватись від класу `Node` і реалізувати, як мінімум, метод `render`. Даному методу передають контекст.

Метод `render` може виконувати будь-які необхідні дії. Все, що поверне даний метод буде вставлене в шаблон на місці тегу, за який він відповідає. Якщо даний тег нічого не вставляє, тоді він має повернати порожню стрічку. Також даний тег має по максимуму ловити усі помилки і не показувати їх на зовні користувачеві. Звичайно, при неправильно переданих аргументах можна і

⁴⁰⁸<http://djbook.ru/rel1.7/howto/custom-template-tags.html#registering-the-tag>

потрібно виводити помилку, щоб розробник одразу міг це побачити та виправити.

Отже, продовжуємо роботу із модулем pagenav.py:

Реалізація етапу виконання

```
1 class PageNavNode(template.Node):
2
3     def __init__(self, object_list, is_paginated, paginator):
4         self.object_list = template.Variable(object_list)
5         self.is_paginated = template.Variable(is_paginated)
6         self.paginator = template.Variable(paginator)
7
8     def render(self, context):
9         t = template.loader.get_template('students/pagination.html')
10        return t.render(template.Context({
11            'object_list': self.object_list.resolve(context),
12            'is_paginated': self.is_paginated.resolve(context),
13            'paginator': self.paginator.resolve(context)},
14        ))
```

Тут важливо пройтись по усіх рядках:

- 1ий рядок: декларуємо клас PageNavNode і унаслідуємось від Node;
- 3ій: також декларуємо метод `init`, щоб зберегти отримані аргументи в об'єкт нашого класу; пізніше ми скористаємося цими даними в методі `render`;
- 4ий: зверніть увагу, що шаблон передає нашому тегу не статичні стрічки, а змінні; а отримуємо ми їх у функції компілятора в якості стрічок; тому на етапі `render` тегу нам прийдеться самостійно відтворити значення даних змінних в контексті шаблона; для цього уже на етапі створення об'єкта PageNavNode ми зберігаємо не просто стрічки (назви змінних), а огортаємо їх в об'єкт `Variable`, який пізніше дозволить нам отримати значення даної змінної в контексті основного шаблона;

- 8ий: починаємо наш основний метод нода render; він отримує об'єкт context; це той самий контекст, який ми з вами розбирали в главі про в'юшки; контекст містить усі деталі основного шаблону і допоможе отримати значення переданих змінних в контексті шаблону;
- 9ий: ця стрічка також повинна бути вам знайома; вона завантажує шаблон pagination.html і повертає відповідний об'єкт класу Template; далі ми скористаємось ним, щоб “відрендерити” (від слова render) кінцевий HTML код із Django шаблона;
- 10ий: викликаємо метод render шаблона; при цьому передаємо йому об'єкт контекста; з допомогою Context об'єкта ми можемо шаблону передати ряд змінних;
- 11ий: тут ми передаємо шаблону pagination.html змінну object_list; при цьому її значення отримуємо із об'єкта Variable, скориставшись методом resolve; даний метод спробує віднайти значення змінної object_list в основному шаблоні; як він має доступ до нього? - через об'єкт контексту переданий в якості аргумента методу render (аргумент context).

Коду насправді не багато. Проте він цікавий тим, що ми власноручно завантажуємо шаблони і динамізуємо шаблонні змінні. Метод render віddaє повністю виконаний шаблон pagination.html із переданими йому змінними.

Також маємо оновити код в шаблоні students_list.html, який віdpovідає за вкладення pagination.html шаблону. Замість нього скористаємось тепер тегом pagenav і не забуваємо попередньо завантажити даний тег на початку шаблону:

Оновлюємо students_list.html

```
1  {% extends "students/base.html" %} 
2  {% load i18n %} 
3  {% load static from staticfiles %} 
4  {% load pagenav %} 
5 
6  ... 
7 
8  {% block content %}
```

```
9  
10 ...  
11  
12 {% pagenav students is_paginated paginator %}  
13  
14 {% endblock content %}
```

Спробуйте тепер перевантажити ваш Django сервер, а тоді головну сторінку із списком студентів. Якщо все зроблено правильно, тоді ви не повинні побачити жодних змін. Посторінкова навігація повинна працювати як працювала:



Посторінкова навігація в дії

На самостійне невелике завдання вам потрібно скористатись новим тегом в усіх шаблонах, де у нас є посторінкова навігація.

Простий тег (simple tag)

Ми щойно реалізували доволі простий тег. Але, тим не менше, код і загальна інфраструктура, що складається з двох етапів, не є на стільки тривіальною.

В нашому випадку ми маємо доволі простий функціонал:

- приймаємо ряд змінних;
- повертаємо HTML код, що згенерувався з врахуванням переданих змінних.

Даний функціонал є доволі типовим для більшості проектів і тому в Django реалізували більш прості способи для його досягнення. В даній секції розглянемо спрощений варіант нашого тегу. Тобто функціонал буде такий самий, але код для його реалізації значно простішим.

Фреймворк дає нам функцію `simple_tag`⁴⁰⁹, яка дозволить у кілька рядочків переписати наш тег `pagenav`.

Замість декорування функції компіляції `pagenav` декоратором `tag`, ми скористаємося функцією `simple_tag` в якості декоратора. Ось як виглядатиме оновлений код тега:

Наш тег з використанням `simple_tag`

```

1 # Usage: {% pagenav object_list=students is_paginated=is_paginated p \
2 aginator=paginator %}
3
4 @register.simple_tag
5 def pagenav(*args, **kwargs):
6     t = template.loader.get_template('students/pagination.html')
7     return t.render(template.Context({
8         'object_list': kwargs['object_list'],
9         'is_paginated': kwargs['is_paginated'],
10        'paginator': kwargs['paginator']})
11    ))
```

По-суті, уся логіка вмістилась в єдину функцію. Данна функція, завдяки декоратору `simple_tag`, отримує усі позиційні та ключові аргументи передані тегу. При цьому передані дані уже не є стрічками, а обчисленими значеннями змінних. Тобто нам не треба користуватись класом `Variable`, щоб перетворити назви змінних у їхні значення.

Зауважте також, що ми покращили синтаксис самого тегу, що можете бачити із першого рядка з документацією. Тепер ми передаємо усі змінні тегу в якості ключових аргументів. Тому порядок не важливий, а важлива лише назва змінної. У попередньому підході нам для цього знадобилося б більше маніпуляцій із стрічками. А тут `simple_tag` уже все підготував і вклав в аргумент `kwargs`.

⁴⁰⁹<http://djbook.ru/rel1.7/howto/custom-template-tags.html#simple-tags>

Також не забудьте оновити код тегу pagenav, адже тепер він працює із ключовими аргументами:

Оновлений тег pagenav в шаблоні students_list.html

```
1  {% pagenav object_list=students is_paginated=is_paginated paginator=\n2  paginator %}
```

Збережіть новий код в модулі pagenav.py, шаблоні students_list.html і переконайтесь, що посторінкова навігація працює і далі без змін.

Тег включення (inclusion tag)

На завершення огляду кастомних тегів розглянемо ще одну допоміжну функцію, яка ідеально підходить для реалізації нашого тегу pagenav.

Ця функція називається inclusion_tag і служить для швидкої реалізації тегів, що вкладають на сторінку код згенерований іншими шаблонами.

Єдина різниця даної функції від simple_tag це те, що вона зробить за нас роботу підготовки і виконання шаблону. Тобто кінцевий код буде ще меншим. Отже, ще один варіант кастомного тегу pagenav:

Наш тег з використанням inclusion_tag

```
1  # Usage: {% pagenav object_list=students is_paginated=is_paginated p\
2  aginator=paginator %}\n\n3\n4\n5  @register.inclusion_tag('students/pagination.html')\n6  def pagenav(object_list, is_paginated, paginator):\n7      """Display page navigation for given list of objects"""\n8      return {\n9          'object_list': object_list,\n10         'is_paginated': is_paginated,\n11         'paginator': paginator\n12     }
```

Як бачите, тепер ми декоруємо функцію pagenav у inclusion_tag. При цьому передали йому шлях до шаблону pagination.html. Код функції максимально простий: повертаємо словник із даними необхідними для шаблону pagination.html. Решту роботи за нас зробить inclusion_tag.

Саме ж використання тегу pagenav в шаблоні залишається незмінним. inclusion_tag також робить за нас розбір усіх ключових аргументів тегу.

Спробуйте тепер ще раз оновити домашню сторінку аплікації у браузері і переконайтесь, що посторінкова навігація і далі працює без змін.

...

Також з допомогою тегів можна додавати **нові змінні⁴¹⁰** в контекст шаблону. Крім того, можна реалізувати тег, який не буде виводив результат прямо в шаблон, а присвоював його **певній змінній⁴¹¹**. Але це все вам на домашню роботу.

Як згадано вище, реалізуйте ще два варіанти тегу pagenav та його вставки у шаблон: через змінну в контексті шаблона, яка встановлюється безпосередньо кастомним тегом та через присвоєння результату виконання тегу змінній (з допомогою ключового слова as).

Після глави про в'юшки ви мали б динамізувати вибір поточної закладки веб-аплікації. Тобто, якщо ви знаходитесь на сторінці Групи, тоді закладка Групи підсвічена як выбрана. Пропоную розробити ще один власний тег, який міститиме логіку з визначення поточної закладки і вставлятиме текст “class=”active”” на закладку, якщо вона зараз обрана. Ось наш початковий код:

⁴¹⁰<http://djbook.ru/rel1.7/howto/custom-template-tags.html#setting-a-variable-in-the-context>

⁴¹¹<http://djbook.ru/rel1.7/howto/custom-template-tags.html#assignment-tags>

Закладка Групи

```
1 <li role="presentation" {% if '/groups' in request.path %}class="active"\n2   ><a href="{% url "groups" %}">{% trans "Groups" %}</a>\n3 ></li>
```

Він має переїхати в тег select_menu, який єдиним параметром приймає назву URL шаблону, а повертає код для класу active або порожню стрічку:

Кастомний тег select_menu

```
1 <li role="presentation" {% select_menu 'groups' %}><a href="{% url "\\\n2 groups" %}">{% trans "Groups" %}</a></li>
```

І на завершення, знайдіть в коді Django фреймворка код тегу for і спробуйте із ним розібратись. Побачите, що код доволі складний. Тому не поспішайте із результатом. Розтягніть задоволення на кілька днів ;-) Подібного роду завдання працюють на перспективу вашого росту.

На цьому завершуємо із шаблонними тегами і переходимо до шаблонних фільтрів:

Кастомний фільтр

Останнім нашим завданням даної глави буде розробка власного шаблонного фільтра.

Фільтр є значно простішою компонентою, ніж шаблонний тег. Фільтр представляє собою звичайну Python функцію, яка приймає один або два аргументи: значення змінної та значення аргументу (опціональний аргумент).

Фільтри також йдуть в папку templatetags і реєструються подібним чином до тегів. Але фільтри реєструємо із допомогою функції “register.filter”.

Фільтр str2int

В першій версії шаблону форми додавання студента `students_add.html` ми застосували фільтр “`add:0`”, щоб перевести стрічку в число. Це нам знадобилось в полі вибору групи, щоб умовний оператор `if` спрацював коректно і отримав обидва операнди одинакового типу: ціле число.

Поле вибору групи на формі додавання студента `students_add.html`

```

1 <select name="student_group" id="student_group"
2         class="form-control">
3     <option value="">{% trans "Select group" %}</option>
4     {% for group in groups %}
5         <option value="{{ group.id }}" {% if group.id == request.POS\
6 T.student_group|add:0 %}selected="1"{% endif %}>{{ group.title }}</o\
7 ption>
8     {% endfor %}
9 </select>

```

Даний підхід є хаком і кращим варіантом буде використання фільтра, що переведитиме стрічку у ціле число. Цим ми і займемось.

Назовемо даний фільтр `str2int` (string to integer) і створимо для нього одноіменний Python модуль в папці `templatetags`:

Створюємо модуль `str2int.py`

```

1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students/temp1\
2 atetags
3 $ touch str2int.py

```

Відкриваємо новостворений модуль в редакторі коду і імплементуємо функцію `str2int`:

Код фільтра str2int

```

1 from django import template
2
3 register = template.Library()
4
5 @register.filter
6 def str2int(string):
7     """Convert input string into integer. If can not convert, return\
8     0"""
9     try:
10         value = int(string)
11     except ValueError:
12         value = 0
13     return value

```

Функцію str2int декоруємо функцією “register.filter”. Даний декоратор реєструє нашу функцію в якості шаблонного фільтра. Код функції доволі простий. В ньому ми конвертуємо отриману стрічку в ціле число. Якщо це зробити неможливо і виникає помилка, тоді повертаємо з нашого фільтра нуль.

Тепер ми готові скористатись новоствореним фільтром в шаблоні:

Оновлюємо шаблон studens_add.html

```

1 {% extends "students/base.html" %} 
2
3 {% load i18n %} 
4 {% load static from staticfiles %} 
5 {% load str2int %} 
6
7 ...
8
9     {% for group in groups %} 
10         <option value="{{ group.id }}" {% if group.id == request.POS\ 
11 T.student_group|str2int %}selected="1"{% endif %}>{{ group.title }}<\ 

```

```
12 </option>
13     {% endfor %}
14
15 ...
```

Перевантажте Django сервер і оновіть форму додавання студента. Виберіть одну із доступних груп в полі Група і не заповнюйте решту полів. Запостіть форму. Отримаєте форму із помилками при валідації. При цьому поле вибору групи має зберегти попередньо обрану вами групу. Якщо так, тоді ви все зробили правильно. Якщо ні, тоді перегляньте ваші попередні зміни і спробуйте самостійно розібратись.

...

Цього мінімуму вам буде достатньо, щоб розуміти принцип функціонування шаблонних фільтрів. Як бачите, там все доволі просто. Якщо знаєте як писати Python функції, тоді знаєте як писати Django фільтри для шаблонів.

На домашнє завдання пропоную вам реалізувати фільтр, який отримуватиме змінну - об'єкт користувача (User), а повертаємо його повне ім'я або користувацьке ім'я (username), якщо поля Ім'я та Прізвище є порожніми. Назвіть даний фільтр nice_username. А тоді скористайтесь ним у шаблонах, де маємо подібні теги:

Виведення повного ім'я користувача

```
1 {{ user.get_full_name|default:user.username }}

---


```

Ось як виглядатиме даний код після застосування фільтру nice_username.
Трохи лаконічніше:

Використання фільтру nice_username

```
1  {% user|nice_username %}
```

Домашнє завдання

Ви вже зауважили, що глави, які йдуть після глав про в'юшки, форми та моделі, є значно простішими. Як я і обіцяв, якщо справились із трьома основними главами, тоді з іншими точно проблем не буде.

Дана глава також не мала б бути виключенням для вас. Матеріалу не багато і він, навіть беручи до уваги усі домашки, є порівняно не складний.

Ми з вами базово розібрали поняття мілдвари, Django команди, шаблонного тегу та фільтра. Навчилися реалізувати не складні кастомні компоненти.

...

Ви вже отримали достатньо домашніх завдань протягом даної глави. В якості невеликого сюрпризу і нагороди за пророблену важку роботу, і за те, що пройшли да даного місяця книги, додаткових домашніх завдань в даній главі давати вам не буду!

Ви молодець! Маю велику надію, що хоча б 50% домашок попередніх глав є зробленими.

А якщо ні, тоді завжди маєте можливість повернутись і спробувати їх зробити ще раз!

...

У наступній главі ми візьмемо доволі важливу і непросту тему. Розберемось із автоматичними тестами. Наприкінці цієї глави матимете відповідь на наступні запитання:

- Що таке юніт тести?

- Для чого потрібні автоматичні тести?
- Що потрібно тестувати в проекті?
- Як правильно тестувати різні компоненти в Django проекті?

...

I не забувайте нагороджувати себе好好ою паузою між главами. Ніщо так не додає бажання продовжувати навчання та роботу як гарний відпочинок!

14. Автоматичні тести: покриваємо тестами код аплікації

На даний момент ми з вами реалізували увесь запланований код по проекту База Студентів. Ця і наступні глави уже не принесуть додатковий функціонал до нашої веб-аплікації.

Проте нам залишилось зробити ще кілька важливих речей. Однією із них є реалізація автоматичних тестів, які покриють код проекту і доведуть, що там немає місця критичним помилкам.

В цій главі ми розберемось із поняттями тесту, покриття коду тестами, видами тестів. Також розглянемо коли варто і коли не варто писати тести.

На практиці навчимось покривати тестами різноманітні компоненти Django веб-аплікації: моделі, в'юшки, форми, обробник сигналу, команду, кастомний фільтр і т.д.

В даній главі ми працюватимемо лише над автоматизованими функціональними тестами на стороні сервера. Тести клієнтської сторони та складні фреймворки типу [Robot Framework⁴¹²](#) залишаються поза межами даної книги.

Почнемо, як завжди, із теоретичного вступу. А далі перейдемо до практики та написання тестів для нашої аплікації `students`.

Що таке тести, коли їх писати та для чого?

Існує велика кількість типів тестів. Відповідно до типу тести можуть мати різне призначення:

⁴¹²<http://robotframework.org>

- довести, що код проекту працює коректно;
- вказати на існуючу помилку в коді програми;
- перевірити правильність конфігурації системи у різних середовищах (демонстраційне, розробницьке, продакшин);
- перевірити безпеку програми;
- показати, що програма готова до відповідних навантажень з боку користувачів;
- перевірити, що функціонал аплікації є легким у користуванні;
- допомогти спроектувати структуру та дизайн проекту.

Таким чином, *тест* - це автоматизована (тест виконує код) чи ручна (тест виконує людина) дія, яка, використовуючи відповідний набір інструментів, виконує одне із вищеперелічених завдань. Найбільш поширеним завданням є перевірка коду програми на коректність і відсутність помилок.

В даній главі ми зосередимось саме на автоматизованих тестиах, які перевіряють коректність раніше написаного нами коду веб-аплікації. Усі інші типи тестів ми зачіпати не будемо і вони залишаються поза даною главою.

Цього вам буде абсолютно достатньо, щоб пройти співбесіду. Також, якщо ви уже програмуєте, але ще не знайомі із створенням автоматичних тестів, саме ці тести (на перевірку коректності коду) є основними тестами, з яких варто почати знайомство із наукою тестування для програміста.

Типи тестів

Існує велика кількість критеріїв для розподілу тестів на типи. Кожен із типів має свій ідеальний час та причину для створення. Давайте коротенько проідемось по категоріях. В кожній категорії наведемо мету, час для написання в циклі проекту та причину для створення даного типу тесту.

По об'єкту тестування:

- **Функціональне тестування⁴¹³**: тести на функціонал програми; створюються до, протягом або після написання коду програми; мають зміст

⁴¹³<http://bit.ly/vpfunctest>

у довготривалих проектах для підвищення якості коду та стабільності роботи аплікації;

- **тестування продуктивності⁴¹⁴:** сюди входять тести по навантаженню програми, стрес-тестування та тестування стабільності програми; дані тести необхідні для гарантії достатньої продуктивності роботи програми при заданій кількості користувачів; дані тести не варто писати на початковому етапі життя аплікації, коли користувачів мало; зазвичай, дані тести створюються пізніше, якщо кількість користувачів різко збільшується;
- конфігураційне тестування: дані тести допомагають перевірити правильність налаштування системи у різних середовищах; актуальні при розробці програми, яка використовується на різних платформах, середовищах та пристроях;
- **юзабіліті тестування⁴¹⁵:** з англ. usability - ергономічність, зручність; ці тести перевіряють функціонал на зручність для користувача; зазвичай, дані тести проводяться з допомогою залучення реальних користувачів в якості тестувальників;
- тестування інтерфейса користувача: тест, що перевіряє чи користувальський інтерфейс має запланований вигляд; наприклад, у веб-розробці це будуть тести, які будуть перевіряти розміри та розташування елементів на веб-сторінці; так, на даний момент існують інструменти, що дозволяють тестувати навіть верстку у вебі;
- **тестування безпеки⁴¹⁶:** тести для оцінки вразливості системи до різного роду атак; створюються в особливо критичних частинах програм (платіжна система, програма бек-офіс банку, і т.д.), а також при виявленні дірки в безпеці;
- тестування локалізації: тести, які перевіряють переклад програми на одну із мов; дані тести доцільно проводити в багатомовних програмах при кожному оновленні перекладів;
- тестування сумісності: перевірка окремих компонент програми на сумісність між собою; також перевірка зовнішнього програмного забезпечення для коректної роботи програми в їхньому середовищі.

⁴¹⁴<http://bit.ly/vptestprod>

⁴¹⁵<http://bit.ly/vpusabtest>

⁴¹⁶<http://bit.ly/vpsectest>

По *доступу до системи*:

- **тестування чорного ящика**⁴¹⁷: тестування програми з точки зору кінцевого користувача, при якому не використовуються знання про внутрішню будову програми;
- **тестування білого ящика**⁴¹⁸: тестування самого коду з метою перевірки коректності роботи програми; зазвичай, методами тієї мови, на якій писалась сама програма; більшість тестів, які ми створюватимемо в даній главі будуть саме тестами білого ящика;
- тестування сірого ящика: це тестування чорного ящика при цьому знаючи внутрішню сторону програми; такі знання допомагають краще організувати самі тести і покривати більшу область можливих дефектів у програмі.

По *рівню автоматизації*:

- **ручне тестування**⁴¹⁹: тести, які проводять тестувальники без використання програмних засобів;
- **автоматизоване тестування**⁴²⁰: дані тести створюються один раз і далі використовуються при кожній потребі в процесі життя програми; дані тести використовують програмні засоби для виконання тестів і перевірки результатів;
- **напівавтоматизоване тестування**: в даному випадку частина тесту відбувається вручну, а інша частина з допомогою наперед підготовленого коду.

По *рівню ізольованості компонентів*:

- **модульне тестування**⁴²¹: також називається юніт тестуванням (з англ. unit - модуль, одиниця); тести, що дозволяють перевірити на коректність

⁴¹⁷<http://bit.ly/vpblacktest>

⁴¹⁸<http://bit.ly/vpwhitetest>

⁴¹⁹<http://bit.ly/vpmantest>

⁴²⁰<http://bit.ly/vpautotest>

⁴²¹<http://bit.ly/vpunittest>

окрему одиницю коду (функцію, клас і т.д.); при цьому тести пишуться так, щоб предмет тесту був максимально незалежним від інших компонент в системі;

- **інтеграційне тестування⁴²²:** тестування модулів об'єднаних в групи; більшість тестів у даній главі будуть саме інтеграційними, адже у фреймворку Django більшість компонент зав'язані на базу даних і цілу низку інших аспектів, які унеможливлюють тестування окремих модулів коду;
- **системне тестування⁴²³:** тестування програми як цілісної системи включно з платформою, на якій її запускають; мета такого тестування виявити усі помилки, що стосуються як функціоналу програми, так і конфліктів сумісності із кінцевою платформою запуску програми.

По *періоду запуску тестів*:

- **альфа-тестування⁴²⁴:** проводиться на ранньому етапі розробки програми; зазвичай проводять потенційні користувачі або замовники;
- **бета-тестування⁴²⁵:** проводиться на кінцевому етапі розробки програми; зазвичай проводить невелика група реальних користувачів.

Багато типів тестів перетинаються між собою у вищеперелічені категоріях. Так, наприклад, тестування білого та чорного ящика зазвичай є функціональними тестами та системними тестами. Тест чорного ящика можна проводити як вручну, так і з допомогою коду.

В даній главі ми працюватимемо із автоматизованими тестами. Тестування проводитиметься здебільшого у вигляді тестування білого та сірого ящиків. Буде лише кілька юніт тестів, а решту належатиме до тестів інтеграційних в силу того, що ми працюємо всередині фреймворку, а не пишемо код з нуля. Усі тести можна сміливо буде назвати функціональними тестами. Також буде один-два тести на безпеку, а саме на перевірку доступу для аноніма до кількох в'юшок нашої аплікації. Інших типів тестів (продуктивність, навантаженість, конфігурація і т.д.) ми не розглядатимемо в даній главі.

⁴²²<http://bit.ly/vpintegtest>

⁴²³<http://bit.ly/vpsystest>

⁴²⁴<http://bit.ly/vpalfatest>

⁴²⁵<http://bit.ly/vpbetatetest>

Писати тести чи ні?

Тепер, коли знаємо, що тести бувають різних типів і, відповідно, їхня мета створення відрізняється, можемо розібратись, як визначати чи потрібні нам тести в програмі чи ні?

Більшість типів тестів (такі як тести на продуктивність, зручність у користуванні, конфігураційних, на сумісність і т.д.), крім функціональних тестів, не варто писати одразу на старті проекту. Потреба у них з'явиться згодом. Велика частина проектів, де ви будете задіяні, взагалі не потребуватиме подібного роду тестів.

В даній главі ми писатимемо функціональні тести. Питання про доцільність тестів даного типу ми і розглянемо детальніше в даній секції.

Серед розробників є ревні прихильники тестів, які стверджують, що тести потрібно писати завжди і покривати тестами варто максимальну кількість коду в проекті. Також є і ті, хто ніколи не приділяє часу автоматичним тестам.

Я, як завжди, рекомендую шукати золоту середину між двома таборами. Кожен проект варто розглядати окремо на предмет наступних питань:

- Писати тести чи ні?
- Коли саме починати роботу над ними?
- Як багато варто додавати тестів у проект?

А взагалі, для чого нам потрібні функціональні тести?

Автоматизовані функціональні тести перевіряють код програми на коректну роботу. Ключове слово “автоматизовані” означає, що розробник чи адміністратор може сам проганяти їх, коли це необхідно. Без допомоги ззовні від тестера чи будь-кого іншого.

Таким чином, функціональні тести дозволяють швидко перевірити чи новий код розробника працює та чи не поламав старий функціонал. Також вони дозволяють зробити швидкий тест перед тим як закидати новий реліз програми на кінцевий сервер. Тести допомагають швидше виявити винуватця помилки по історії комітів в репозиторії коду. Тести допомагають уникати очевидних

людських помилок. Функціональні тести підвищують загальний рівень якості коду.

Також тести служать документацією для складного коду і часто допомагають програмісту розібратись у чужому коді. Крім того, при рефакторингу існуючого коду можна легко виявити і відремонтувати поламаний раніше працюючий код. Функціональні тести також дозволяють швидко протестувати програму на різних платформах і версіях середовища.

Після усіх вищеперечислених плюсів функціональних автоматизованих тестів складається враження, що їх варто писати завжди. Проте, після того, як я почав працювати напряму із клієнтами та обговорювати бізнес цілі проекту, а не лише його технічну сторону, почав розуміти, що тести - це не лише користь, але й додатковий бюджет для замовника.

Автоматизовані тести - це завжди код, на який треба витратити додатковий час. Часом навіть 30-50% від загального часу проекту, взалежності від його складності та кількості задіяних зовнішніх компонент та сервісів. Відповідно, на цей процент зростає вартість проекту. Крім того, тести - це річ, яка потребує часу на подальшу підтримку та оновлення відповідно до змін специфікацій проекту.

Так. Тести - це інвестиція в майбутнє проекту. З цим не можна не погодитись. Саме з цього місця і постають наступні запитання:

- Якого роду проект? Розовий чи довготерміновий?
- Яке можливе майбутнє проекту? Невизначене і з'ясується по-ходу діла?
А чи він точно буде затребуваним у найближчі кілька років?

Взалежності від відповідей на вищенаведені запитання потреба в тестах буде різною. Ось кілька правил, якими я керуюсь, коли рекомендую або ж, навпаки, не раджу працювати над тестами в проекті:

- якщо проект невеликий і разовий (тобто подальші розробки будуть мінімальними і не буде глобального рефакторингу коду), тоді можна обійтись без тестів; в такому проекті вони будуть не інвестицією, а витратою часу та коштів;

- якщо великий і, планується, довготривалий проект пишеться з нуля та при цьому вся бізнес модель (ідея) не протестована, тоді проект, швидше за все, потребуватиме частих та кардинальних експериментів у функціоналі; тоді тести не варто писати спочатку; лише на фазі, коли бізнес модель запрацювала і вже є достатньо даних, щоб зафіксувати основні специфікації функціоналу та покрити їх тести; в протилежному випадку, якщо одразу почати писати тести в такому проекті, тести можуть переписуватись з нуля (так само як і код самого проекту) по кілька разів;
- в попередньому випадку (великий і довготривалий проект), якщо проект має хороше фінансування (із запасом) і точно потребуватиме неодноразового рефакторингу, варто одразу писати тести, але лише тести чорного ящика; в такому випадку, навіть повністю змінивши код проекту, розробник матиме швидший доказ того, що кінцевий функціонал не постраждав;
- якщо проект пишеться вже під тестовану бізнес ідею, яка працює і є чітко визначені потреби, що не змінююватимуться, тоді тести також можна сміливо писати одразу; в майбутньому вони лише поверталися дивіденди у вигляді зекономлених годин на баг-фікси.

Ну і взагалі, ваша справа, як технічного спеціаліста, надати вищій ланці проекту усі плюси та мінуси, “за” і “проти” тестів саме в рамках даного проекту. І вже або ви разом, або вища ланка самостійно, вирішує доцільність тести. Головне, щоб в кінці кінців тести виявилися у колонці інвестицій, а не в колонці витрат.

...

Тепер по періоду написання тести в житті проекту. Є кілька варіантів, коли варто починати роботу над тести:

- тести пишемо ще до написання самого коду програми; так зване TDD (Test Driven Development, детальне пояснення у наступній секції);
- тести пишемо паралельно до написання нового коду;

- тести пишемо наприкінці проекту вже коли узгоджені кінцеві специфікації, які могли змінюватись в процесі розробки;
- є ще такий тип тесту, як тест на знайдену помилку; відповідно, його додають, якщо в існуючому функціоналі програми на кінцевому середовищі (сервері) знайдено помилку.

Загалом, краще наповнювати проект тестами раніше, ніж пізніше. Але, знову ж таки, все залежить від бізнес цілей проекту і від запитань, що ми розглянули вище про доцільність написання тестів.

Варіант написання тестів до написання самого коду ми розглянемо пізніше. Він має свої плюси та мінуси. Він не особливо відрізняється від написання тестів паралельно до написання коду з точки зору бюджету проекту.

А от написання тестів лише після тесту бізнес ідеї і затвердження остаточного функціоналу проекту в першому релізі може мати свій великий плюс: економія часу на кількаразове переписування існуючих тестів.

Підсумовуючи: в проекті із точними кінцевими специфікаціями можна одразу приступати до створення тестів. В іншому випадку, краще зекономити трохи часу і коштів та додати тести після того, як функціонал програми “устакани-ться”.

Якими повинні бути функціональні тести?

При написанні тестів варто пам'ятати кілька важливих моментів, які допоможуть вам зберегти трохи часу та підвищити їхню якість:

- краще, коли один тест тестує лише одну річ (функцію, клас, блок коду);
- 100% покриття коду тестами не гарантує відсутності помилок;
- з попереднього пункту випливає, що більше тестів не означає кращий результат; краще написати кілька тестів, що покривають критичний функціонал програми;
- також з певного моменту збільшення кількості тестів стає малоефективним; тому варто в проекті додати лише найбільш необхідні тести; 20% тестів дозволяють виявити 80% помилок;

- таким чином, не варто намагатись покривати увесь код і кожний умовний оператор тестами; краще використайте цей час на розробку нового функціоналу.

...

Наприкінці теоретичної частини даної глави розглянемо ще кілька популярних англомовних абревіатур і слів у світі тестування:

TDD, Continuous Integration, Code Test Coverage

В цій теоретичній секції розберемось із трьома термінами, які наведені вище у заголовку. Останнім із них (Test Coverage) ми навіть скористаємося наприкінці даної глави.

TDD

Test Driven Development (TDD⁴²⁶) - розробка програмного забезпечення через написання тестів.

При такому підході програміст спочатку пише тест на не існуючий функціонал чи зміну до існуючого функціоналу. Спочатку тест має провалюватись. Далі програміст додає відповідний код і після цього написані ним тести проходять успішно.

Переваги TDD підходу:

- в результаті TDD підходу отримуємо дійсно якісні тести, а не просто галочку про їх наявність;
- TDD дозволяє краще продумати архітектуру програми ще на етапі написання тестів.

Недоліки TDD підходу:

- для початківця такий підхід забере більше часу, ніж написання тестів на вже існуючий код;

⁴²⁶<http://bit.ly/vptddtest>

- при зміні специфікацій проекту прийдеться витратити додатковий час на оновлення тестів також.

TDD підхід можна і варто використовувати у всіх проектах, в яких було вирішено писати код одразу з тестами. Даний підхід підвищить якість коду проекту і зменшить кількість помилок. Звісно, в деяких випадках TDD може забирати ще більше часу, ніж написання тестів після написання коду. Тут, знову ж таки, варто обирати підхід відповідно до бізнес-цілей проекту, його бюджету і часових рамок.

Continuous Integration

Continuous Integration (з англ. безперервна інтеграція⁴²⁷) - це практика розробки програмного забезпечення, при якій проводяться регулярні збірки коду проекту з метою виявлення помилок інтеграції.

Дана практика полягає у автоматизації наступних процесів:

- отримання коду проекту з репозиторія;
- збірка проекту на окремому сервері, в середовищі максимально наближеному до кінцевого;
- запуск тестів;
- розгортання готового проекту;
- та підготовка звітів у тому чи іншому вигляді.

Вище описана процедура може виконуватись, наприклад, раз на добу. Або при кожному пуші (коміті) розробника на віддалений сервер. Таким чином, можна легко побачити, що після коміту одного із членів розробницької команди тести програми поламались. Швидко виявити дану проблему та віправити.

Тобто можливість постійно моніторити стан системи на наявність помилок і є основною задачею практики Безперервної Інтеграції (Continuous Integration).

На практиці, щоб реалізувати даний підхід, використовують один із існуючих інструментів. Ось невеликий список одних із найпопулярніших опен-

⁴²⁷<http://bit.ly/vpcinteg>

сорсних систем Безперервної Інтеграції: [Hudson](#)⁴²⁸, [Jenkins](#)⁴²⁹, [CruiseControl](#)⁴³⁰, [CruiseControl.NET](#)⁴³¹.

Ну і, звичайно, код проекту обов'язково повинен знаходитись в репозиторії коду, щоб була можливість реалізувати дану практику.

Code Test Coverage

Code Test Coverage - [рівень покриття коду](#)⁴³² програми тестами. Код вважається *покритим*, якщо він був запущеним протягом прогонки тестів.

Покриття коду тестами прийнято вважати показником якості коду проекту. Хоча його не можна вважати однозначним лічильником. 100% покриття коду тестами ще не означає, що код не містить помилок. Адже кількість тестів ще не означає, що самі тести є якісні.

Відповідно, не варто намагатись покривати ваш код на всі 100% тестами. Зазвичай, достатньо написати кілька тестів, що покриватимуть лише найбільш критичні місця програми.

Якщо розглядати Django проект, то достатньо факту, що кожна в'юшка, форма і інша кастомна компонента була запущена в результаті прогонки тестів. І не обов'язково домагатись варіанту, коли ваш тест змусить інтерпретатор увійти у кожен умовний оператор в коді. Результат від затрачених годин на такі тести є мізерним.

Django спільнота пропонує свої інструменти щодо перевірки покриття коду проекту тестами. Ми скористаємося одним із них наприкінці даної глави.

Бібліотека unittest

В цій секції коротко оглянемо як на практиці виглядають тести і ті терміни, з якими пов'язані автоматичні тести у світі Python.

Мова Python надає тестовий фреймворк [unittest](#)⁴³³, який також називають PyUnit. Це версія бібліотеки JUnit в мові Java, яка, у свою чергу, є версією

⁴²⁸<http://hudson-ci.org/>

⁴²⁹<https://jenkins-ci.org/>

⁴³⁰<https://en.wikipedia.org/wiki/CruiseControl>

⁴³¹<http://www.cruisecontrolnet.org/>

⁴³²<http://bit.ly/vpcov>

⁴³³<https://docs.python.org/2/library/unittest.html>

тестової бібліотеки у мові Smalltalk. Кожна із даних бібліотек є стандартом тестового фреймворка у світі своєї мови.

Бібліотека unittest підтримує автоматизовані тести надаючи інструменти для налаштування тестового середовища, збірки тестів в колекції, їхній запуск та організацію звітів про виконані тести. Щоб реалізувати усі ці компоненти, unittest підтримує ряд певних важливих концепцій:

- *test fixture* (тестова фікстура) - набір підготовочних дій для прогонки одного або кількох тестів, що потребують одинакових налаштувань тестового середовища;
- *test case* - Python клас, що збирає докупи тестові методи; тестовий кейс вважається найменшою одиницею тестування; unittest надає базовий клас для унаслідування у своїх тестових класах;
- *test suite* (з англ. suite - комплект) - колекція тестових кейсів або навіть самих сюйтів; тестовий сюйт збирає тести, що повинні запускатись разом;
- *test runner* - компонента, яка збирає, запускає та видає результат прогонки тестів.

На практиці ми матимемо справу лише з класами, що представлятимуть тестові кейси (Test Case). Решту тестових аспектів та компонент від нас прихована фреймворк Django. В більшості випадків нам навіть не потрібно до кінця розуміти як все відбувається всередині тестових процесів.

Таким чином, на кожен функціонал і набір відповідних функцій, класів та методів у нашій аплікації ми створюватимемо окремий тестовий клас. Даний клас повинен містити тестові методи, назва кожного із яких починається префіксом ‘test’. Усі інші назви методів ігноруються тестовим фреймворком при запуску тестів.

Всередині тестового методу ми маємо набір унаслідуваних методів, які дозволяють тестувати потрібний результат функціоналу під питанням. Одним із основних таких методів є `assertEqual`, який перевіряє чи передані аргументи є рівними.

Давайте розглянемо коротенький приклад тестового кейсу:

Приклад тестового кейсу і методів

```
1 import unittest
2
3 class TestStringMethods(unittest.TestCase):
4
5     def test_upper(self):
6         self.assertEqual('apple'.upper(), 'APPLE')
7
8     def test_isupper(self):
9         self.assertTrue('ORANGE'.isupper())
10        self.assertFalse('Orange'.isupper())
11
12    def test_split(self):
13        s = 'hello world'
14        self.assertEqual(s.split(), ['hello', 'world'])
15        # check that s.split fails when the separator is not a string
16        self.assertRaises(TypeError, s.split, 2)
```

Вищеприведений приклад взятий з офіційної документації мови Python. Давайте детальніше глянемо на код:

- 1ий рядок: імпортуємо модуль unittest, який надасть нам усі необхідні компоненти для розробки тестів;
- 3ий: декларуємо тестовий клас; він збиратиме тестові методи, що об'єднуватимуться у єдину фікстуру і тестуватимуть спільний функціонал; клас тестового кейсу повинен унаслідуватись від класу TestCase; у фреймворку Django є свій TestCase, але про нього поговоримо пізніше; даний клас збирає методи, що тестують поведінку будованого класу стрічки в мові Python;
- 5ий: тестовий метод обов'язково повинен починатись із стрічки ‘test’; окрім аргументу self він нічого не приймає; задача кожного із тестового методу - відзвітити чи він пройшов успішно, а чи поламався;
- 6ий: в даному тривіальному методі test_upper ми перевіряємо чи метод стрічки upper повертає правильне значення; робимо це з допомогою

одного із найчастіше використовуваних тестових (перевірочних) методів, що дає нам клас TestCase: assertEquals; даному методу ми передаємо два значення і він або реєструє помилку, якщо передані йому значення не є рівними, або тихо пропускає код йти далі; це і є основною сутністю автоматичних тестів;

- 8ий: в даному тестовому методі ми перевіряємо на коректність інший метод стрічки - isupper; зауважте, що у цьому методі ми використовуємо перевірочні методи більше, ніж один раз; при цьому тестовий фреймворк зарахує усі ці перевірки, як один тест; тобто один тестовий метод відповідає одному тесту у тестових звітах; до формату звітів ми дійдемо згодом;
- 9ий: тут ми використовуємо інший тестовий метод під назвою assertTrue; він перевіряє чи отримане значення є правдивим і відповідно до результату перевірки реєструє помилку або успіх для тестових звітів;
- 10ий: подібним чином тільки з точністю до навпаки працює перевірочний метод assertFalse; стрічка ‘Foo’ не містить усіх символів, які були б у вищому реестрі; саме тому виклик isupper поверне значення False; а віддак assertFalse зареєструє дану перевірку як успішну; тобто тест пройде;
- 12ий: в останньому тестовому методі буде продемонстрований ще один цікавий і популярний метод;
- 14ий: тут ми використовуємо вже знайомий нам метод assertEquals; в даному випадку ми перевіряємо метод стрічки split; в деталі обидвох значень даного виклику вдаватись не будемо; на даний момент це повинно бути для вас доволі очевидним; в протилежному випадку “загугліть” принцип роботи методу split;
- 16ий: метод assertRaises перевіряє чи передана йому функція (або інший об’єкт, який можна викликати) дійсно поверне помилку заданого типу; в нашому випадку ми методу split передаємо аргумент 2, який не задовільняє тип аргументу, який даний метод може обробити; split першим аргументом отримує стрічку для поділу основної стрічки на частини; а ми спробували передати йому число і, відповідно, повинні отримати помилку типу - TypeError; таким чином, метод assertRaises приймає першим аргументом клас типу помилку, другим - функцію, клас чи метод для виклику, а під рештою аргументів - позиційні та

ключові аргументи для другого аргументу.

Таким чином, визначивши Python клас із набором тестових методів ми зможемо підготувати автоматизовані тести на потрібний нам функціонал.

Перевірочних методів типу `assertEqual` є значно більше, ніж ми розібрали у прикладі вище. Крім того, Django фреймворк надає свої додаткові методи. Ті з них, якими користуватимемось на практиці, розбератимемо далі.

Крім тестових методів клас тестового кейсу може містити спеціально підготовані методи, що впливають на тестове середовище. Давайте оглянемо ще один приклад:

Методи впливу на налаштування тестового середовища

```
1 import unittest
2
3 class DemoTestCase(unittest.TestCase):
4
5     def setUp(self):
6         self.student = Student()
7         self.student.save()
8
9     def tearDown(self):
10        self.student.delete()
11        self.student = None
12
13     @classmethod
14     def setUpClass(cls):
15        cls.db_connection = createDBConnection()
16
17     @classmethod
18     def tearDownClass(cls):
19        cls.db_connection.destroy()
20        cls.db_connection = None
21
22     def test_something(self):
23        ...
```

У вищезгаданому прикладі ми бачимо Python клас із чотирма методами, які не починаються стрічкою ‘test’. Вони призначені для налаштування тестового середовища і запускаються тестовим фреймворком у різні моменти прогонки тестів. Давайте детальніше розберемо, коли саме вони запускаються і з якими цілями:

- 5ий рядок: метод setUp містить код, який запускатиметься тестовим фреймворком перед кожним тестовим методом класу тест кейса; setUp може бути корисним для таких дій, як набивка тестових даних; кожен із тестових методів впливатиме на ці дані і тому перед кожним тестовим методом ці дані потрібно приводити у початковий стан;
- 9ий: подібним чином до setUp метод tearDown запускається при запуску кожного тестовому метода; але він викликається *після* його запуску; таким чином в методі tearDown зазвичай видаляють усі налаштування зроблені в методі setUp;
- 14ий: метод класу setUpClass містить код, який запускається лише один раз для цілого тестового класу, перед початком запуску тестових методів даного класу; зазвичай використовується для налаштувань та даних, що повинні зберігати свій стан і не змінюватись протягом запуску усіх тестів даного класу; це може бути створення доступу до бази даних, отримання даних із зовнішніх ресурсів чи інші дії, що є ресурсозатратними; даний метод є методом класу, а не інстансу, тому ми його декоруємо в функцію `classmethod`⁴³⁴;
- 18ий: подібним чином метод tearDownClass запускається лише один раз після прогонки усіх тестових методів даного класу; зазвичай у ньому ми відключаємо і видаляємо усі дані та налаштування пророблені в методі setUpClass.

Ми часто використовуватимемо методи setUp та tearDown у подальших тестах. Іншими методами практично не користуватимемось.

Також для запуску тестів потрібно передати дані тестові класи тестовим сюїтам (test suite), щоб тестовий фреймворк їх розпізнав. Проте ми не розглянемо подібний запуск в середовищі unittest, адже використовуватимемо

⁴³⁴<https://docs.python.org/2/library/functions.html#classmethod>

тестове середовище Django. А Django фреймворк розпізнає для нас усі наші тести по назвах модулів та тестових методах. Без додаткових налаштувань.

...

На цьому ми завершуємо теоретичний огляд поняття тестування коду. Більше про тестування, його види і цілі ви можете почерпнути із онлайн ресурсу ProTesting⁴³⁵.

Ми готові переходити до практики. А почнемо із розгляду інструментарія із запуску тестів в середовищі Django проекту:

Готуємо тестовий фреймворк

Протягом даної глави ми писатимемо тести для коду, що знаходиться в Django аплікації students. На даний момент можемо бачити модуль tests.py в корені цієї аплікації.

При запуску тестів тестовий фреймворк шукає усі тести, що знаходяться всередині модулів, чиї назви починаються із слова test. Тестів у нас буде багато і дописувати ми будемо їх у різні модулі, тому краще підготувати для них окрему папку. Назовемо її tests і покладемо в корені аплікації students:

Готуємо підпакет для тестів

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students
2
3 # видаляємо старий тестовий модуль, він нам не знадобиться
4 $ rm tests.py
5
6 # створюємо папку для майбутніх тестів
7 $ mkdir tests
8
9 # перетворюємо просту папку в Python пакет
10 $ cd tests
11 $ touch __init__.py
```

⁴³⁵<http://www.protesting.ru>

В даній папці далі ми будемо створювати Python модулі під назвою із префіксом test.

Для тестового середовища Django фреймворк кожного разу, при запуску тестів, створює окрему тестову базу даних. Щоб її створювати фреймворк повинен мати доступ до сервера бази даних від імені користувача, який має право створювати нову базу даних. Тестова база даних необхідна для того, щоб кожного разу керувати даними з чистого листка та не зачіпати даних у справжній базі даних. Таким чином, тести не ламають даних ні на вашому розробницькому середовищі, ні на кінцевому чи демо серверах.

Можна самостійно створити тестову базу даних і вказати дані доступу до неї у налаштуваннях проекту. Також можна надати повний доступ включно з дозволом створювати нову базу даних нашому існуючому MySQL користувачу students_db_user. Ну і найпростіший варіант: можна просто у власному розробницькому середовищі змінити поточного користувача в налаштуваннях DATABASES на кореневого MySQL адміністратора.

Ми скористаємось найпростішим варіантом і скористаємось рутовим (root) користувачем MySQL, якого створили ще на самому початку інсталяції та налаштування сервера баз даних. Відкриваємо модуль db.py в корені проекту і робимо наступні зміни:

Налаштування тестової бази даних в модулі db.py

```
1 DATABASES = {
2     'default': {
3         'ENGINE': 'django.db.backends.mysql',
4         'HOST': 'localhost',
5         # 'USER': 'students_db_user',
6         # 'PASSWORD': 'password',
7         'USER': 'root',
8         'PASSWORD': 'root_password',
9         'NAME': 'students_db',
10        'TEST': {
11            'CHARSET': 'utf8',
12            'COLLATION': 'utf8_general_ci',
13        }
14    }
15}
```

```
14      }
15 }
```

Давайте розберемо оновлені рядки коду:

- 5ий рядок: ми закоментували деталі користувача `students_db_user`, який має доступ лише до існуючої бази нашого проекту;
- 8ий: натомість ключ ‘USER’ тепер вказує на користувача `root`; даного користувача ми створили з вами протягом глави книги, коли інсталювали і конфігурували MySQL базу даних;
- 9ий: тут вам потрібно вставити пароль користувача `root`, який ви встановили при інсталляції MySQL;
- 10ий: ми додали новий ключ під іменем ‘TEST’; він вказує на ще один словник, що містить деталі налаштування саме тестової бази даних; в даному словнику не обов’язково дублювати усі ключі основної бази даних (під назвою ‘`default`’), а лише достатньо додати ті ключі, що потрібно вказати для тестової бази; усі інші налаштування автоматично унаслідуються із основної бази;
- 11ий: для тестової бази даних нам також необіхдне кодування `utf8`, адже нам потрібно, щоб база вміла коректно працювати із кириличними символами.

Тепер у нас все готово, щоб спробувати перший раз запустити тести:

Перший запуск тестів

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb
2 (studentsdb)VipodLaptop:studentsdb vipod$ python manage.py test stud\ents
3
4 Creating test database for alias 'default'...
5
6 -----
7 --
8 Ran 0 tests in 0.000s
9
```

```
10 OK
11 Destroying test database for alias 'default'...
```

Як бачите, ми знову скористались скриптом `manage.py`. Цього разу його командою `test`. Йй ми передали назву аплікації, в якій потрібно шукати тести. Звісно, було запущено 0 тестів, адже у нас їх поки немає. Також можете бачити в останньому рядку виводу команди `test` повідомлення про тестову базу даних.

Команда `test` дозволяє запускати тести в окремих модулях, тестових класах та навіть окремі тестові методи. Деталі форматування таких аргументів можете переглянути із документації даної команди:

Варіанти запуску команди `test`

```
1 (studentsdb)VipodLaptop:studentsdb vipod$ python manage.py test --he\
2 lp
3 Usage: manage.py test [options] [path.to.modulename|path.to.modulena\
4 me.TestCase|path.to.modulename.TestCase.test_method] ...
5
6 Discover and run tests in the specified modules or the current direc\
7 tory.
```

Далі, при написанні тестів ми використовуватимемо дану можливість команди `test`, щоб проганяти лише ті тести, що ми писали в тій чи іншій секції.

Тепер ми повністю готові, щоб переходити до написання самих тестів.

Тест утиліти

Першим ділом попрацюємо над покриттям тестами коду модуля `util.py`. В цьому модулі у нас є три функції:

Вміст модуля util.py

```
1 from django.core.paginator import Paginator, EmptyPage, PageNotAnInt\
2 eger
3
4 def paginate(objects, size, request, context, var_name='object_list'\
5 ):
6     """Paginate objects provided by view"""
7     # apply pagination
8     paginator = Paginator(objects, size)
9
10    ...
11
12    return context
13
14 def get_groups(request):
15     """Returns list of existing groups"""
16     # deferred import of Group model to avoid cycled imports
17     from .models import Group
18
19    ...
20
21    return groups
22
23 def get_current_group(request):
24     """Returns currently selected group or None"""
25     # we remember selected group in a cookie
26     pk = request.COOKIES.get('current_group')
27
28     if pk:
29         ...
30     else:
31         return None
```

Вище наведено скорочений вигляд трьох функцій:

- paginate: функція отримує список елементів для пагінації і повертає шаблонний контекст наповнений необхідними змінами для організації посторінкової навігації;
- get_groups: функція повертає список груп з бази даних;
- get_current_group: дана функція повертає поточно обрану групу на користувачькому інтерфейсі.

В даній секції ми протестуємо одну функцію: get_current_group. А дві інші функції залишаються вам на домашнє завдання. Код, що йде із даною книгою містить тест на усі функції модуля util. Тому ви зможете порівняти ваші тести і, при потребі, покращити їх.

Усі тести даного модуля покладемо у модуль test_util.py всередині пакету tests:

Готуємо модуль для тестів util.py

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students/tests  
2 $ touch test_util.py
```

Префікс test в назві модуля є обов'язковим, щоб тестовий фреймворк ідентифікував наші тести. Другу частину назви файла ми намагаємось тримати аналогічно до назви модуля, в якому знаходиться код, який ми тестиємо. Це допомагає доволі швидко ідентифікувати місцезнаходження тих чи інших тестів в пакеті tests.

Відкриваємо модуль test_util і додаємо заготовочний код класу тестового кейсу:

Клас-заготовка для тестів функцій з модуля util.py

```
1 from django.test import TestCase
2
3
4 class UtilsTestCase(TestCase):
5     """Test functions from util module"""
6
7     def setUp(self):
8         # create set of users and groups in database
9
10    def test_get_current_group(self):
11        pass
```

Давайте детальніше глянемо на основні моменти даної заготовки:

- 1ий рядок: TestCase тепер використовуємо не з модуля unittest, а з фреймворку Django; даний TestCase надає нам більше можливостей для перевірки, а також робить необхідні налаштування середовища (базу даних, тестовий сервер і т.д.); без нього ми можемо писати лише юніт тести, що абсолютно не залежать від Django проекту;
- 4ий: клас тестового кейсу назовемо UtilsTestCase, адже в ньому ми тестуватимемо функції, що знаходяться в модулі util; звичайно, унаслідуємось від TestCase базового класу;
- 7ий: вже знайомий нам метод setUp; в ньому ми створюватимемо тестовий контент в базі Django проекту; більшість функцій в модулі util оперують над Django моделями і для їхнього тесту потрібно підготувати набір даних;
- 10ий: заготовка методу, що тестуватиме функцію get_current_group.

Код доволі простий і повинен бути зрозумілим, оскільки ми вже розібрали більшість аспектів тестового кейсу в теоретичній частині даної глави.

Якщо ви в даний момент запустите тести, то отримаєте у звіті результат із одним пройденим тестом, адже тестовий метод поки порожній.

Давайте пригадаємо як працює функція get_current_group. Це нам необхідно, щоб зрозуміти, які дані потрібно підготувати для тесту даної функції:

Функція get_current_group

```
1 def get_current_group(request):
2     """Returns currently selected group or None"""
3
4     # we remember selected group in a cookie
5     pk = request.COOKIES.get('current_group')
6
7     if pk:
8         from .models import Group
9         try:
10             group = Group.objects.get(pk=int(pk))
11         except Group.DoesNotExist:
12             return None
13         else:
14             return group
15     else:
16         return None
```

Розбирати її ще раз порядково не будемо, адже робили це під час роботи над даною функцією. Основною задачею даної функції є отримати із запиту куку під назвою current_group і повернути об'єкт групи під ID, що знаходиться в даній куці. Якщо куки немає або ID не дійсне, тоді функція повертає порожнє значення None.

Для повного тесту даної групи нам потрібно в тестовій базі мати хоча б одну групу. Тому в метод setUp додамо код, що готове дану групу.

Саму ж функцію get_current_group тестуватимемо за кількома сценаріями:

- коли кука в запиті порожня;
- коли кука містить ID не існуючої групи;
- і на завершення, коли кука містить ID існуючої групи.

Таким чином ми зможемо покрити основні моменти функціоналу даної функції:

Тестуємо функцію get_groups

```
1 from django.test import TestCase
2 from django.http import HttpRequest
3
4 from students.models import Student, Group
5 from students.util import get_groups
6
7
8 class UtilsTestCase(TestCase):
9     """"Test functions from util module"""
10
11     def setUp(self):
12         # create group
13         group1, created = Group.objects.get_or_create(
14             id=1,
15             title="Group1")
16
17     def test_get_current_group(self):
18         # prepare request object to pass to utility function
19         request = HttpRequest()
20
21         # test with no group set in cookie
22         request.COOKIES['current_group'] = ''
23         self.assertEqual(None, get_current_group(request))
24
25         # test with invalid group id
26         request.COOKIES['current_group'] = '12345'
27         self.assertEqual(None, get_current_group(request))
28
29         # test with proper group identifier
30         group = Group.objects.filter(title='Group1')[0]
31         request.COOKIES['current_group'] = str(group.id)
32         self.assertEqual(group, get_current_group(request))
```

Давайте детально пройдемось по нашему первому завершенному тесту:

- 13ий рядок: в методі setUp ми створюємо групу під назвою Group1; зауважте, що дана група буде створюватися перед кожним запуском іншого тестового методу; а вбудований функціонал тестового фреймворку Django зачищатиме усі дані з бази після кожного тестового методу; тому реалізувати метод tearDown, в якому видаляти створену групу - не потрібно;
- 19ий: код тестового методу функції get_current_group ми розпочинаємо із створення об'єкту запиту; функція get_current_group отримує в якості єдиного аргументу об'єкт запиту і отримує з нього потрібні дані;
- 22ий: єдині дані необхідні в запиті - це кука під назвою 'current_group'; об'єкт запиту має атрибут COOKIES, в якому зберігає словник усіх кук, що прийшли із веб-браузера; в даному рядку ми самостійно встановлюємо значення куки 'current_group' у порожню стрічку; таким чином тестуємо випадок, коли куки немає;
- 23ий: в даному рядку ми перевіряємо чи виклик функції get_current_group при переданому їй запиті із порожньою кукою поверне нам порожнє значення; тестова функція assertEquals ідеально підходить; якщо функція поверне інше значення, ніж None тестовий фреймворк зарахує помилку, яка буде виведена в кінцевому звіті після прогонки усіх тестів;
- 26ий: подібним чином ми встановлюємо значення куки у ID неіснуючої групи;
- 27ий: в цьому випадку наша функція get_current_group повинна повернути порожнє значення;
- 30ий; і в останньому тестовому випадку ми хочемо перевірити чи get_current_group спрацює коректно для выбраної існуючої групи; для цього ми витягуємо заготовану наперед групу Group1 і зберігаємо її у змінну для подальшого використання;
- 31ий: встановлюємо куку в ID знайденої групи;
- 32ий: переконуємось, що функція повернула об'єкт моделі нашої групи.

Як бачите, ідея тестів є доволі простою. Запускаємо функцію із різними вхідними параметрами та перевіряємо чи вона віддає коректні дані у кожному з випадків. Це і є основною суттю автоматизованих тестів.

Ми готові, щоб запустити наш перший реальний тест:

Запускаємо тест функцій модуля util.py

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb
2 # не забудьте активувати віртуальне середовище
3 $ python manage.py test students.tests.test_util
4 Creating test database for alias 'default'...
5 INFO 2015-08-31 12:11:35,328 signals: Student added: Vitaliy Podoba \
6 (ID: 1)
7 .
8 -----
9 --
10 Ran 1 test in 0.066s
11
12 OK
13 Destroying test database for alias 'default'...
```

Ми запустили лише тести, що знаходяться в модулі `students.tests.test_util`. Можете бачити, що тестовий фреймворк створив тестову базу, яку наприкінці тестів видалив.

“INFO” рядок був надрукований нашим кастомним сигналом, що слідкує за операціями над студентами. Бачимо, що протягом прогонки тестів було створено одного студента із ID 1. Після даного рядка бачимо одну крапочку. Кожна крапочка означає один пройдений тест (всередині це один тестовий метод тестового кейса). У випадку, якщо тест не проходить, на місці крапки матимемо літеру F, що означає FAIL (тест не пройдено).

Далі бачимо рядок звіту, що вказує про кількість запущених, пройдених та по-ламаних тестів. А також час, витрачений на запуск тестів. У нашому випадку маємо один єдиний тест і він пройшов вдало. Тому загальний результат OK.

Цієї інформації вам буде достатньо, щоб аналізувати звіт після прогонки більшості автоматизований тестів написаних на основі unittest бібліотеки.

На домашнє завдання вам залишається написати тести на дві інші функції модуля util.py. Коли це зробите, гляньте в код, що йде з книгою і порівняйте власні тести із тими, що там знайдете.

А ми рухаємось до трохи складніших тестів. Тестів, де нам доведеться задіяти тестовий клієнт. З його допомогою отримуватимемо результат дії в'юшок та форм.

Тест форми та в'юшки

Для перевірки компонент, що генерують цілі сторінки (такі як в'юшки та форми) тестовий фреймворк Django надає спеціальний тестовий клієнт. Клієнт - це Python клас із непоганим набором перевірочних методів. З його допомогою можна зробити запит на певну адресу Django веб-проекту, отримати та перевірити вміст відповіді. В цьому і полягає суть тесту в'юшки.

Звичайно, можна власним кодом імпортувати клас чи функцію в'юшки, створити тестовий запит, викликати в'юшку із цим запитом в якості аргумента і перевірити результат виклику. В цьому випадку матимемо тест типу білого ящика. Правда він буде складніший у реалізації і вимагатиме трохи більше коду.

З іншої сторони Django уже дає нам клієнт, який не лише перевірить виклик в'юшки, але й дозволить впевнитись, що в'юшка працює під потрібною веб-адресою. Тобто ми також зможемо перевірити правильність URL шаблонів. При написанні таких тестів ми також заглядатимемо в код в'юшки, щоб розуміти як її краще потестувати і перевірити крайні випадки для вхідних даних. Саме тому можемо назвати ці тести тестами сірого ящика.

Почнемо із покриття тестами коду в'юшки, що відповідає за список студентів.

Тестуємо в'юшку списку студентів

В даній секції ми покриємо тестами в'юшку-функцію students_list, що знаходиться в модулі views/students.py. Данна функція відповідає за в'юшку до-

машньої сторінки. Основною її задачею є вивести список студентів з бази у потрібному форматі.

Почнемо, як завжди, із заготовки. Для цього створимо новий тестовий модуль `test_student_views.py` і підготуємо код тестового класу:

Готуємо тестовий клас для списку студентів

```
1 from datetime import datetime
2
3 from django.test import TestCase, Client
4 from django.core.urlresolvers import reverse
5
6 from students.models import Student, Group
7
8
9 class TestStudentList(TestCase):
10
11     def setUp(self):
12         # create groups
13         group1, created = Group.objects.get_or_create(
14             title="MтM-1")
15
16         ...
17
18         # remember test browser
19         self.client = Client()
20
21         # remember url to our homepage
22         self.url = reverse('home')
23
24     def test_students_list(self):
25         pass
```

Давайте розглянемо кілька цікавих нам рядків:

- 1ий: `datetime` ми поки імпортнули на майбутнє;

- Зій: тут ми імпортуємо клас Client; з його допомогою робитимемо запити на потрібну нам в'юшку 'home' (домашня сторінка, список студентів);
- 9ий: тестовий клас назовемо TestStudentList; він міститиме тестові методи, які покриють усі необхідні випадки в'юшки-функції students_list;
- 13ий: в методі setUp ми, знову ж таки, створюватимемо в базі тестовий контент (групи та студентів);
- 19ий: крім того, в цьому методі заготуємо кілька атрибутів, до яких зможемо пізніше звертатись із коду тестових методів; щоб не писати код для створювання об'єкта клієнта в кожному тестовому методі, ми напишемо його лише один раз на рівні метода setUp і збережемо як атрибут об'єкту тестового кейсту TestStudentList; в кожному із тестових методів будемо до нього звертатись;
- 22ий: подібним чином формуємо веб-адресу до домашньої сторінки нашого проекту і зберігаємо її в атрибуті url;
- 24ий: в наступних кроках ми заповнимо кодом даний тестовий метод.

У проектах, де немає багато в'юшок і тести на усі в'юшки не займатимуть надто багато місця, усі тести можна зібрати в один модуль. Такий модуль зазвичай називають test_views.py. Проте у нашому випадку ми вирішили створювати новий тестовий модуль на кожну групу в'юшок (в'юшки студентів, груп, журналу і т.д.). В кожній групі в нас мінімум по 3 в'юшки, а отже краще розмежувати в'юшки різних моделей у різні модулі.

Давайте нагадаємо нашу стару функцію, що представляла список студентів на домашній сторінці. Зауважте, що навіть, якщо на даний момент ваша в'юшка, що відповідає за список студентів, є переписана на в'юшку у вигляді класу, то повернати її у старий вигляд не потрібно. Ми тестуватимемо її з допомогою тесту сірого ящика, а для нього немає значення внутрішній код програми. Ми скористаємося об'єктом тестового клієнта (замінника веб-браузера у реальному середовищі), який абстрагує нас від Python коду проекту.

Таким чином, до того моменту, поки функціонал домашньої сторінки ви зберегли при переписуванні її на інший Python код (функцію, клас і т.д.), наші

тести працюватимуть одинаково. В цьому і перевага тестів чорного та сірого ящиків.

Отже, ось як виглядає код нашої в'юшки списку студентів:

В'юшка списку студентів

```
1 def students_list(request):
2     # check if we need to show only one group of students
3     current_group = get_current_group(request)
4     if current_group:
5         students = Student.objects.filter(student_group=current_grou\
6 p)
7     else:
8         # otherwise show all students
9         students = Student.objects.all()
10
11    # try to order students list
12    order_by = request.GET.get('order_by', '')
13    if order_by in ('last_name', 'first_name', 'ticket'):
14        students = students.order_by(order_by)
15        if request.GET.get('reverse', '') == '1':
16            students = students.reverse()
17
18    # apply pagination, 3 students per page
19    context = paginate(students, 3, request, {},
20                      var_name='students')
21
22    return render(request, 'students/students_list.html', context)
```

В ідеалі нам варто перевірити основні моменти даної функції:

- звичайний список студентів без будь-яких додаткових умов;
- список з обраною групою;
- список посортований по одному з полів студента;
- список розбитий на підсторінки (pagination).

Ми можемо розділити код тестів на кожен із вищеперелічених аспектів функції у окремі тестові методи, або вкласти весь тестовий код в один метод. Оберемо перший варіант, оскільки деякий функціонал вимагатиме кількох перевірок і, якщо вкладемо усі тести в один метод, то він виявиться занадто громіздким. Крім того матимемо у звітах кілька пройдених тестів, а не один. Відповідно, і розбирати де поломка буде простіше у структурованих тестах.

Спочатку завершимо набивку тестового контенту в методі `setUp` і реалізуємо перший тестовий метод. В `setUp` дописуємо код, що створить наступну структуру даних в базі:

- дві групи ‘MtM-1’ та ‘MtM-2’; в кожну з них ми додамо студентів;
- чотири студента; одного додамо в групу ‘MtM-1’, а решту студентів увійдуть у групу ‘MtM-2’.

В тестовому ж методі ми перевіримо чи звичайний список студентів працює правильним чином:

Перший тестовий метод списку студентів

```
1 from datetime import datetime
2
3 from django.test import TestCase, Client
4 from django.core.urlresolvers import reverse
5
6 from students.models import Student, Group
7
8
9 class TestStudentList(TestCase):
10
11     def setUp(self):
12         # create 2 groups
13         group1, created = Group.objects.get_or_create(
14             title="MtM-1")
15         group2, created = Group.objects.get_or_create(
16             title="MtM-2")
```

```
17
18     # create 4 students: 1 for group1 and 3 for group2
19     Student.objects.get_or_create(
20         first_name="Vitaliy",
21         last_name="Podoba",
22         birthday=datetime.today(),
23         ticket='12345',
24         student_group=group1)
25     Student.objects.get_or_create(
26         first_name="John",
27         last_name="Dobson",
28         birthday=datetime.today(),
29         ticket='23456',
30         student_group=group2)
31     Student.objects.get_or_create(
32         first_name="Sam",
33         last_name="Stefenson",
34         birthday=datetime.today(),
35         ticket='34567',
36         student_group=group2)
37     Student.objects.get_or_create(
38         first_name="Arnold",
39         last_name="Kidney",
40         birthday=datetime.today(),
41         ticket='45678',
42         student_group=group2)
43
44     # remember test browser
45     self.client = Client()
46
47     # remember url to our homepage
48     self.url = reverse('home')
49
50     def test_students_list(self):
51         # make request to the server to get homepage page
```

```
52     response = self.client.get(self.url)
53
54     # have we received OK status from the server?
55     self.assertEqual(response.status_code, 200)
56
57     # do we have student name on a page?
58     self.assertIn('Vitaliy', response.content)
59
60     # do we have link to student edit form?
61     self.assertIn(reverse('students_edit',
62                         kwargs={'pk': Student.objects.all()[0].id}),
63                  response.content)
64
65     # ensure we got 3 students, pagination limit is 3
66     self.assertEqual(len(response.context['students']), 3)
```

Імпорти усі залишилися ті самі, тому одразу переходимо до методу setUp:

- 15ий рядок: створюємо другу групу в базі; обидві групи запам'ятуємо в змінних, щоб далі в методіскористатись для розподілу студентів по групах; метод get_or_create повертає два значення: створений об'єкт і булеванівську змінну, що означає об'єкт був знайдений в базі чи новостворений;
- 19ий: створюємо першого студента і призначаємо його в першу групу;
- 37ий: аналогічним чином набиваємо усіх чотирьох студентів в базі;
- 50ий: в даному методі тестиємо список студентів без додаткових параметрів;
- 52ий: об'єкт клієнта (client) має набір корисних для нас методів та атрибути; одними з важливих є методи для виконання запиту: get, post; кожен із даних методів робить відповідний його назві тип запиту на сервер; в даному рядку ми робимо запит методом GET на сервер за адресою домашньої сторінки нашого проекту; даний виклик повертає відповідь (response) із сервера; це повноцінна відповідь із набором заголовків та тілом, що відображається у браузері; запам'ятуємо відповідь у змінну для подальших тестів;

- 55ий: спочатку переконаємось, що запит відбувся успішно і сервер надіслав нам відповідь із успішним статусним кодом 200; для цього об'єкт відповіді містить атрибут `status_code`;
- 58ий: в якості наступного тесту перевіримо чи ім'я одного із студентів є у відповіді із сервера; вміст відповіді знаходиться в атрибуті `content`; у нашому випадку це текст формату HTML; зауважте, що тут ми скористались новим перевірочним методом `assertIn`; він приходить до нас із самого фреймворку Django, а не бібліотеки unittest; даний метод перевіряє чи передана стрічка знаходиться у HTML тексті; якщо студент Vitaliy знаходиться серед списку студентів, тоді помилки під час запуску даного тесту не виникне;
- 61ий: в цьому рядку ми тестиємо чи на сторінці є лінк до форми редактування першого в списку студента; ми скористались функцією `reverse`, щоб сформувати веб-адресу форми редактування першого в базі студента; і перевіряємо далі чи дана стрічка (веб-адреса) знаходиться на домашній сторінці;
- 66ий: на завершення тестового методу ми перевіримо чи список студентів складається із трьох об'єктів; окрім того, що ми маємо можливість тестиувати заголовки та тіло відповіді, ми також маємо доступ до деяких внутрішніх об'єктів, що передаються з з'юшки у шаблон; усі змінні, що передаються у шаблон з допомогою контекста шаблона попадають в об'єкт відповіді в атрибут `context`; таким чином список студентів є доступним у вигляді `"response.context['students']"`; це дозволяє детально перевірити список студентів; ми ж лише перевіримо чи кількість є коректною; їх має бути 3, адже посторінкова навігація групуює по 3 студента на сторінку; далі ми розберемось окремо із сценарієм посторінкової навігації.

Спробуйте тепер запустити самостійно тести, що знаходяться в модулі `test_student_views.py`. По аналогії до того, як ми це робили із тестами функцій утиліт у попередній секції.

На даний момент ми протестували лише перший сценарій роботи списку студентів. Нам ще залишається перевірити чи коректно працюють сортування, вибір групи та посторінкова навігація студентів.

Кожен із даних сценаріїв вкладемо у окремі тестові методи. Наступним тестовим методом ми зможемо перевірити, що вибір групи впливає на список студентів:

Тест вибраної групи

```
1  def test_current_group(self):
2      # set group1 as currently selected group
3      group = Group.objects.filter(title="MtM-1")[0]
4      self.client.cookies['current_group'] = group.id
5
6      # make request to the server to get homepage page
7      response = self.client.get(self.url)
8
9      # in group1 we have only 1 student
10     self.assertEqual(len(response.context['students']), 1)
```

Давайте детальніше розберемо даний тестовий метод:

- Зій рядок: витягуємо з бази групу під назвою MtM-1 і запам'ятовуємо у змінну для подальшого використання;
- 4ий: об'єкт клієнта містить атрибут cookies для зберігання значень куки, що висилатиметься на сервер під час запиту; ми встановлюємо куку 'current_group' у значення ID групи MtM-1;
- 7ий: робимо запит на в'юшку домашньої сторінки; при цьому клієнт відправить на сервер встановлену куку 'current_group';
- 10ий: в методі setUp ми додали єдиного студента до групи MtM-1; тому домашня сторінка повинна повернути лише одного студента при даній обраній групі; знову ж таки користуємося атрибутом context, щоб отримати список студентів, який передає в'юшка шаблону; його довжина повинна бути рівна одиниці.

Подібним чином тестуємо сортування студентів. Наприклад, оберемо сортування студентів по Прізвищу (поле last_name). Для цього нам прийдеться під час запиту на сервер передати параметр order_by і встановити його у значення last_name:

Тестовий метод test_order_by

```
1 def test_order_by(self):
2     # set order by Last Name
3     response = self.client.get(self.url, {'order_by': 'last_name\
4 })  
5  
6     # now check if we got proper order
7     students = response.context['students']
8     self.assertEqual(students[0].last_name, 'Dobson')
9     self.assertEqual(students[1].last_name, 'Kidney')
10    self.assertEqual(students[2].last_name, 'Podoba')
```

В даному тестовому методі ми виконуємо запит на сервер типу GET і при цьому передаємо параметри. Дану дію можемо виконати з допомогою словника, який передаємо методу get другим аргументом. Таким чином ми “просимо” домашню сторінку посортувати студентів по прізвищу. Далі в тестовому методі перевіряємо чи дійсно студенти посортовані по прізвищу згідно англійської (латинської) абетки. Дістаемось до списку студентів, як завжди, через атрибут context об'єкту відповіді (response).

На домашнє завдання вам залишається протестувати домашню сторінку із списком студентів на предмет зворотнього сортування (reverse order) та посторінкової навігації. Ви вже знаєте як передавати параметри разом із GET запитом на сервер, тому не повинно виникнути проблем із передачею параметрів посторінкової навігації.

На цьому ми завершуємо роботу над тестами в'юшкі, що відповідає за домашню сторінку (спісок студентів) і переходимо до тестів форми редагування студента.

Тестуємо форму редагування студента

До цього часу ми в наших тестових класах в методах `setUp` прописували код, що набивав базу даних демо контентом. Контентом, який ми використовували далі в тестових методах. Проте цього разу ми розглянемо менш громіздкий підхід для підготовки тестових даних.

Ми вже з вами знаємо, що таке фікстури даних, як їх створювати та використовувати, щоб легко наповнювати базу даних. В попередніх випадках ми використовували фікстури, щоб набити базу початковими даними для демо інстансу або оновити дані, якщо ми безнадійно їх поламали в процесі нашої розробки та експериментів на локальному розробницькому інстансі. Цього ж разу ми скористаємось подібними файлами у форматі JSON, які служитимуть фікстурами для тестового контенту.

Тестові фікстури, як і інші (розробницькі, демо контент, початковий контент для продакшин систем), заведено класти у підпапку `fixtures` в корені аплікації.

Давайте підготуємо папку та файл для тестових даних:

Готуємо файл для тестових даних

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students
2 $ mkdir fixtures
3 $ cd fixtures
4 $ touch students_test_data.json
```

Тестові дані підготуємо у файлі під назвою `students_test_data.json`. Назва файла немає значення. На дану назву ми засилатимемось із тестових кейсів.

Формат даних між усіма типами фікстур абсолютно одинаковий. Спробуємо перевести всі дані, які ми створювали у попередньому тестовому класі в методі `setUp` у фікстуру:

Тестові дані у вигляді JSON дампу

```
1 [  
2 {  
3     "fields": {  
4         "first_name": "Vitaliy",  
5         "last_name": "Podoba",  
6         "middle_name": "Ivanovych",  
7         "photo": "./podoba.jpg",  
8         "notes": "",  
9         "student_group": 1,  
10        "birthday": "1984-06-17",  
11        "ticket": "000"  
12    },  
13    "model": "students.student",  
14    "pk": 1  
15 },  
16 {  
17     "fields": {  
18         "first_name": "Student1",  
19         "last_name": "Last Name 1",  
20         "middle_name": "Middle Name 1",  
21         "photo": "",  
22         "notes": "Notes 1",  
23         "student_group": 1,  
24         "birthday": "1994-10-19",  
25         "ticket": "111"  
26    },  
27    "model": "students.student",  
28    "pk": 2  
29 },  
30 ...  
31 ...  
32 ...  
33 {  
34     "fields": {
```

```
35         "notes": "",  
36         "leader": 1,  
37         "title": "Group1"  
38     },  
39     "model": "students.group",  
40     "pk": 1  
41 },  
42 {  
43     "fields": {  
44         "notes": "",  
45         "leader": null,  
46         "title": "Group2"  
47     },  
48     "model": "students.group",  
49     "pk": 2  
50 },  
51 {  
52     "fields": {  
53         "username": "admin",  
54         "first_name": "",  
55         "last_name": "",  
56         "is_active": true,  
57         "is_superuser": true,  
58         "is_staff": true,  
59         "last_login": "2014-11-19T00:17:12.803Z",  
60         "groups": [],  
61         "user_permissions": [],  
62         "password": "pbkdf2_sha256$12000$vwDz0jnc05bN$cZSSKK0kBa0RNA\  
63 vEyRqNY10rIfFpH6mp0xdLyc2aB0g=",  
64         "email": "admin@admin.com",  
65         "date_joined": "2014-11-01T12:59:32.011Z"  
66     },  
67     "model": "auth.user",  
68     "pk": 1  
69 }
```

Вище наведено лише фрагмент даного файла. Повну версію можете знайти в коді, що йде із книгою.

Детально розглядати даний файл не будемо, адже ви, швидше за все, уже розумієте як він працює базуючись на матеріалах глави про моделі та бази даних. В даному файлі ми заготовували інформацію для чотирьох студентів та двох груп. Кожного із студентів віднесли до однієї із цих двох груп. Крім того, ми створили адміна, який має права робити усі дії. Під ним ми логуватимемось на сайт і робитимемо дії, які заборонені аноніму.

Тепер можемо скористатись даним файлом у нашому новому тестовому кейсі, що перевірятиме роботу форми редагування студента. Для цього створюємо новий тестовий модуль `test_student_forms.py` і набиваємо заготовочкою з новим тестовим класом:

Клас тестового кейсу для форми редагування студента

```
1 from django.test import TestCase, Client
2 from django.core.urlresolvers import reverse
3
4
5 class TestStudentUpdateForm(TestCase):
6
7     fixtures = ['students_test_data.json']
8
9     def setUp(self):
10         # remember test browser
11         self.client = Client()
12
13         # remember url to edit form
14         self.url = reverse('students_edit', kwargs={'pk': 1})
15
16     def test_form(self):
17         pass
```

Тестовий кейс для форми редагування ми назвали `TestStudentUpdateForm`. В рядку 7 ми задекларували атрибут класу під назвою `fixtures`. Його значення є списком назв файлів, які запускатимуться перед кожним тестовим методом. В нашему випадку список складається із одного елемента під назвою `'students-test_data.json'`. Це файл, який ми напередодні створили і наповнили в папці `fixtures`. Саме з даної папки в корені аплікації `students` тестовий фреймворк автоматично знайде потрібні файли.

Таким чином, замість маси Python коду у кожному методі `setUp` ми один раз створили тестову фікстуру і можемо користуватись нею у всіх тестових кейсах, де нам це необхідно.

В методі `ж setUp` ми створили і запам'ятали об'єкт класу `Client`. А також згенерували і зберегли веб-адресу до форми редагування студента із ID 1. Даний студент буде створений завдяки файлу тестової фікстури.

Ми готові до обговорення самих тестових методів форми редагування студентів. Ось які варіанти можна і варто протестувати на нашій формі:

- наявність елементів на сторінці форми редагування; тобто форма у режимі запиту GET;
- тест спроби збереження даних при правильно заповнених даних; іншими словами тестуємо коректність роботи кнопки Зберегти на формі редагування;
- тест кнопки Скасувати;
- тест на доступ до форми; наприклад, перевірка, що анонім не має права зайти на сторінку із формою редагування;
- також можна потестувати форму на наявність стилів: класі, типів тегів і т.д.

Цього буде абсолютно достатньо, щоб на 90% переконатись, що форма перебуває в робочому стані.

Почнемо із перевірки форми при запиті методом GET. В даному тестовому методі нам необхідно залогуватись, щоб отримати доступ до групи. Далі ми робимо GET запит на сторінку з формою редагування студента із ID 1. Ну і, звичайно, перевіряємо чи форма містить кілька ключових елементів.

Тестуємо форму при методі GET, тобто отримуємо саму форму із сервера

```
1 def test_form(self):
2     # login as admin to access student edit form
3     self.client.login(username='admin', password='admin')
4
5     # get form and check few fields there
6     response = self.client.get(self.url)
7
8     # check response status
9     self.assertEqual(response.status_code, 200)
10
11    # check page title, few field titles and button on edit form
12    self.assertIn('Edit Student', response.content)
13    self.assertIn('Ticket', response.content)
14    self.assertIn('Last Name', response.content)
15    self.assertIn('name="add_button"', response.content)
16    self.assertIn('name="cancel_button"', response.content)
17    self.assertIn('action="%s"' % self.url, response.content)
18    self.assertIn('podoba.jpg', response.content)
```

Порядково глянемо на даний тестовий метод:

- Зій рядок: кліент також має метод `login`, який дозволяє робити запити на сервер в якості залогованого користувача; ми скористалися даним методом, щоб увійти на веб-сайт під користувачем `admin`;
- бий: робимо запит типу GET на сторінку із формою редагування; зберігаємо об'єкт відповіді для подальших тестів;
- 9ий: перевіряємо чи сервер повернув нам успішний статусний код: 200;
- 12ий: перевіряємо чи на сторінці присутній заголовок із стрічкою ‘Edit Student’; в цьому нам допомагає атрибут `content` об'єкта відповіді (`response`);
- 13ий: подібним чином перевіряємо наявність інших полів і атрибутів HTML елементів на сторінці; в основному вибірковим чином і не потрібно пробувати все перевірити;

- 17ий: тут ми перевіряємо чи є атрибут action на сторінці, що вказує на форму редагування студента (тобто поточну сторінку); ми знаємо, що даний атрибут обов'язково має бути в тегу форми, а отже і перевіряємо його присутність.

Спробуйте тепер запустити тести форми редагування студента. Ось що отримаєте:

Не забудьте запустити базу даних перед запуском тестів

```
1 $ python manage.py test students.tests.test_student_forms
2 Creating test database for alias 'default'...
3 INFO 2015-09-04 19:36:54,245 signals: Student added: Vitaliy Podoba \
4 (ID: 1)
5 INFO 2015-09-04 19:36:54,267 signals: Student added: Student1 Last N\
6 ame 1 (ID: 2)
7 INFO 2015-09-04 19:36:54,303 signals: Student added: Student2 Last N\
8 ame 2 (ID: 3)
9 INFO 2015-09-04 19:36:54,307 signals: Student added: Student3 Last N\
10 ame 3 (ID: 4)
11 F
12 =====
13 ==
14 FAIL: test_form (students.tests.test_student_forms.TestStudentUpdate\
15 Form)
16 -----
17 --
18 Traceback (most recent call last):
19   File "/data/work/virtualenvs/studentsdb/src/studentsdb/students/te\
20 sts/test_student_forms.py", line 36, in test_form
21     self.assertIn('Edit Student', response.content)
22 AssertionError: 'Edit Student' not found in '\n\n<!DOCTYPE html>\n<h\
23 tml lang="uk">\n<head>\n  <meta charset="UTF-8"/>\n  <title>\xd0\xaa1\
24 \xd0\xb5\xd1\x80\xd0\xb2\xd1\x96\xd1\x81
```

Наприкінці даного виводу можна бачити помилку AssertionError, яка чітко дає зрозуміти, що 'Edit Student' стрічка не знайдена серед наступного тексту.

І наступний текст містить багато символів типу ‘\xd0\xa1’ і тому подібних. Таким чином я отримав у свою консоль символи, що не є в таблиці ASCII символів. В нашому випадку це кирилиця.

Як і в реальному середовищі, в тестовому варіанті веб-сайт перекладається на українську мову. Нам же ж, для коректних тестів, які не залежать від поточно активованої на сайті мови, потрібно відключити переклади і показувати сторінки в канонічному варіанті. Тобто англійською мовою.

Лише ті тести, що напряму тестиють чи правильно працюють переклади повинні активовувати інші мови, окрім канонічної (в нашому випадку канонічною є англійська, адже нею ми користувались напряму в шаблонах).

Замість того, щоб вручну змінювати мову проекту на англійську в settings.py кожного разу при прогонці тестів, ми скористаємось кращим варіантом. Для можливості змінювати налаштування модуля settings.py Django фреймворк надає декоратор override_settings. З його допомогою ми можемо змінювати налаштування проекту як для окремого тестового метода, так і для цілого тестового кейсу.

У нашому випадку ми змінимо мову проекту для цілого тестового кейсу:

Змінюємо мову проекту

```
1 ...
2 from django.test import TestCase, Client, override_settings
3 ...
4
5 @override_settings(LANGUAGE_CODE='en')
6 class TestStudentUpdateForm(TestCase):
7     ...
```

Ми декорували тестовий клас TestStudentUpdateForm функцією override_settings. Їй ми передали назву налаштування і його бажане значення. У нашому випадку ми встановили змінну LANGUAGE_CODE у англійську мову.

Спробуйте ще раз запустити тести форми редагування:

Успішний запуск тестів із англійською мовою

```
1 $ python manage.py test students.tests.test_student_forms
2 Creating test database for alias 'default'...
3 INFO 2015-09-04 19:45:49,856 signals: Student added: Vitaliy Podoba \
4 (ID: 1)
5 ...
6 .
7 -----
8 --
9 Ran 1 test in 0.704s
10
11 OK
12 Destroying test database for alias 'default'...
```

Цього разу наш, поки єдиний, тестовий метод має успішно пройти.

Тепер давайте протестуємо чи правильно працює обробник форми редагування. Для цього нам треба буде організувати запит типу POST на сервер передавши при цьому усі необхідні дані. Тестовий метод назовемо `test_success` (перевірка успішного посту даних з форми на сервер):

Тестуємо кнопку Зберегти на формі

```
1 def test_success(self):
2     # login as admin to access student edit form
3     self.client.login(username='admin', password='admin')
4
5     # post form with valid data
6     group = Group.objects.filter(title='Group2')[0]
7     response = self.client.post(self.url, {'first_name': 'Update\
8 d Name',
9         'last_name': 'Updated Last Name', 'ticket': '567',
10        'student_group': group.id, 'birthday': '1990-11-11'}, fo\
11 llow=True)
12
```

```
13     # check response status
14     self.assertEqual(response.status_code, 200)
15
16     # test updated student details
17     student = Student.objects.get(pk=1)
18     self.assertEqual(student.first_name, 'Updated Name')
19     self.assertEqual(student.last_name, 'Updated Last Name')
20     self.assertEqual(student.ticket, '567')
21     self.assertEqual(student.student_group, group)
22
23     # check proper redirect after form post
24     self.assertIn('Student updated successfully', response.content)
25 nt)
26     self.assertEqual(response.redirect_chain[0][0],
27         'http://testserver/?status_message=' +
28         'Student%20updated%20successfully!')
```

В даному методі ми тестируємо кілька різних джерел даних: відповідь із сервера, а також об'єкт студента в базі на наявність оновлених полів. Давайте детальніше оглянемо рядки даного методу:

- Зій рядок: логуємось на сайт під адміном;
- бий: дістаємо групу з бази; групу, до якої належить студент із ID 1; нам потрібно буде знати ID даної групи, щоб передати на сервер в якості поля групи студента;
- 7ий: робимо запит на сервер типу POST; метод post клієнта отримує адресу, куди робити пост та набір полів форми у вигляді словника; у словнику передаємо усі обов'язкові поля форми редагування студента такі як Ім'я, Прізвище, Квиток, Група студента та дату народження; також зверніть увагу, що ми передали ще один третій аргумент методу post - follow; з його допомогою ми вказуємо клієнту слідувати редіректам із сервера, якщо такі матимуть місце; якщо пам'ятаєте, при успішній обробці форми редагування студента ми редіректимо користувача на домашню сторінку і показуємо статусне повідомлення про успішність операції; тому важливо, щоб наш тестовий клієнт прослідував згідно

даного редіректу (тобто автоматично зробив другий запит на сервер) і повернув нам об'єкт відповіді (`response`) із головною сторінкою; об'єкт відповіді запам'ятали у змінну `response`, яку використовуватимемо для проведення перевірок;

- 14ий: першим ділом перевіримо статусний код відповіді із сервера; зверніть увагу, що код має бути рівним 200, а не коду редіректа (301 або 302); це тому, що ми передали методу `post` “`follow=True`” аргумент; кліент прослідував згідно редіректу, зробив повторний запит на сервер і у відповідь отримав домашню сторінку із кодом успіху - 200;
- 17ий: витягуємо з бази студента, якого ми щойно оновили з допомогою запиту `POST`; будемо звіряти оновлені дані із бази із тим, що ми передали методу `post`;
- 18ий: перевіряємо чи нове Ім'я студента оновлено; в наступних рядках подібним чином перевіряємо інші поля, які ми змінили напередодні;
- 21ий: в даному рядку перевіряємо чи група студента була оновленою; пам'ятаємо, що в полі зв'язку знаходиться об'єкт відповідного типу моделі; у нашому випадку це модель групи;
- 24ий: із сервера ми отримаємо редірект, а після редіректу клієнт віддасть нам відповідь із головною сторінкою нашої аплікації; під час редіректу в'юшка обробки форми редактування студента також додає статусне повідомлення; у даному рядку ми перевіряємо чи текст даного повідомлення міститься в контенті домашньої сторінки: ‘`Student updated successfully`’;
- 26ий: остання перевірка є перевіркою на веб-адресу самого редіректу; об'єкт відповіді `response` містить атрибут `redirect_chain`; даний атрибут з'явився завдяки нашему аргументу “`follow=True`” під час запиту `post`; він містить список кортежів із двох елементів; кожен кортеж містить інформацію про проміжні веб-адреси і статусні коди HTTP протоколу; під час успішного поста форми на сервер ми отримали один редірект на головну сторінку; таким чином атрибут `redirect_chain` містить лише один кортеж даних; кортеж складається із адреси редіректу та статусного коду; в даному рядку ми звіряємо чи редірект на головну сторінку містить правильну веб-адресу; зверніть увагу, що в тестовому середовищі Django тестовий сервер і аплікація зазвичай працюють під адресою “`http://testserver`”.

Можете знову запустити тести модуля `test_student_forms.py` і переконатись, що наш новий тест проходить успішно.

Зауважте, що ми не висилаємо додаткового поля на сервер, яке підтверджує перевірку на CSRF атаку. У тестовому середовищі Django відключає подібні перевірки, щоб полегшити нам життя.

На домашнє завдання попрошу ваш покращити даний тестовий метод і научитись відсылати на сервер серед інших даних також поле фото студента. Звісно, після успішної відправки треба буде перевірити чи фото коректно збережене.

На завершення тестування форми розглянемо варіант, коли пробуємо зайди на форму під анонімним користувачем. Це буде так званий тест на безпеку.

Для цього реалізуємо ще один тестовий метод і назовемо його `test_access` (тест на доступ):

Тест на права доступу до форми редагування

```
1 def test_access(self):
2     # try to access form as anonymous user
3     response = self.client.get(self.url, follow=True)
4
5     # we have to get 200 code and login form
6     self.assertEqual(response.status_code, 200)
7
8     # check that we're on login form
9     self.assertIn('Login Form', response.content)
10
11    # check redirect url
12    self.assertEqual(response.redirect_chain[0],
```

```
13     ('http://testserver/users/login/?next=/students/1/edit/' \
14 , 302))
```

Як бачите, тест доволі простий. Щоб потрапити на форму під анонімом, ми спеціально не логуємося на сайт. Далі робимо запит GET на сторінку із формою. При цьому знову передаємо аргумент “follow=True”, щоб клієнт слідував можливим редіректам із сервера.

При спробі доступу до ресурсу, до якого користувач немає достатньо прав, Django фреймворк перенаправить (redirect) його на сторінку із формою входу. Саме це і відбулось з нами при запиті на форму редагування студента.

В останніх двох перевірочних викликах ми переконуємося, що потрапили на форму “Login Form”, а прийшли туди з допомогою редіректу на адресу “`../users/login/...`”.

На домашнє завдання вам залишається потестувати кнопку Скасувати на формі редагування. А також можна перевірити наявність потрібних CSS класів та інших елементів в HTML коді, що забезпечують потрібний вигляд форми. Після виконання даного завдання можете звірити власний результат із кодом, що йде із книгою.

Тест моделі

Далеко не всі моделі містять кастомні методи і бізнес логіку. Адже основна задача класу моделі - оголосити набір полів для розміщення даних у базі.

Тому перед тим як одразу переходити до коду, давайте коротенько пройдемось по загальних рекомендаціях стосовно того, який саме код варто покривати тестами.

Що варто тестувати?

Серед розробників немає чітких правил, що потрібно тестувати і що тестувати не варто. Кожен сам визначає свої правила як загальні так і кожен раз вирішу-

ючи в окремих ситуаціях. Також в кожній спільноти мови програмування чи фреймворку можуть існувати свої традиції.

У світі Django зазвичай керуються наступними п'ятьма правилами:

- якщо код є рідним Python кодом (бібліотекою) - його ми не тестиємо;
- якщо код є рідним кодом фреймворку Django (аплікація вбудована в Django) - його ми не тестиємо;
- якщо код належить до сторонньої Django аплікації чи сторонньої Python бібліотеки написаним кимось іншим - його також не тестиємо;
- ваш власний код, що реалізує бізнес логіку аплікації (Django в'юшки, форми, кастомні теги, процесори контексту, мідлвари, команди і т.д.) - тестиувати потрібно;
- код Django моделі тестиється, якщо він містить кастомні методи; тобто логіку.

Таким чином, в більшості випадків, Django модель тестиють лише, якщо вона містить кастомні методи і логіку. Наявність потрібних полів зазвичай не перевіряється.

Наша модель студента містить один кастомний метод, тому переходимо до тесту класу студента.

Тестуємо модель студента

Якщо пригадаємо код моделі студента, то побачимо, що він містить набір полів та метод unicode. Даний метод і буде основною ціллю тестів моделі.

Створюємо черговий тестовий модуль в підпакеті tests:

Готуємо модуль для тестів моделі

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students/tests  
2 $ touch test_models.py
```

Даний модуль ми назвали test_models.py і призначили для тестів усіх моделей. Проте у даній секції ми протестуємо лише модель студента. Тести інших моделей практично ідентичні.

Тестуємо модель студента

```
1 from django.test import TestCase
2
3 from ..models import Student
4
5
6 class StudentModelTests(TestCase):
7     """Test student model"""
8
9     def test_unicode(self):
10         student = Student(first_name='Demo', last_name='Student')
11         self.assertEqual(unicode(student), u'Demo Student')
```

Давайте порядково розберемо код вищеннаведеного модуля:

- 6ий: тестовий клас назвемо StudentModelTests, щоб з назви було зрозуміло його призначення;
- 9ий: єдиний тестовий метод даного класу є метод, що тестиє `__unicode__` - метод класу моделі; відповідно даемо тестовому методу назву `test_unicode`, щоб пізніше можна було легко розуміти його призначення;
- 10ий: для перевірки методу нам потрібен студент, тому створюємо об'єкт студента; нам потрібні лише поля Ім'я та Прізвище для тесту метода `__unicode__`;
- 11ий: в останньому рядку тестового метода перевіряємо чи результат методу `__unicode__` повертає потрібний результат; зауважте, що ми не викликаємо напряму даний метод моделі; натомість користуємося вбудованою Python функцією `unicode`; Python інтерпретатор самостійно звернеться до методу `__unicode__` об'єкта `student` всередині виклику `unicode(student)`.

Код тесту надзвичайно простий, адже тестиємо лише один метод класу моделі студента. В складніших проектах інколи приходиться кастомізувати багато вбудованих методів моделі, щоб впливати на поведінку коду, що працює із

базою даних. Це може бути кастомний менеджер моделей (objects атрибут класу моделі), кастомні поля і налаштування моделі.

В нашому випадку ми обійшлися, як то кажуть, малою кров'ю.

На домашнє завдання попрошу вас покрити тестами решту моделей в аплікації students.

Тест команди

В цій секції покриємо тестами наші кастомні команди. Їх у нас лише одна. Якщо пам'ятаєте, то вона називається `stcount` і виводить у консоль кількість об'єктів різних типів. В залежності від переданих їй аргументів.

Давайте пригадаємо її код:

Код команди `stcount`

```
1 from django.core.management.base import BaseCommand
2 from django.contrib.auth.models import User
3
4 from students.models import Student, Group
5
6
7 class Command(BaseCommand):
8     args = '<model_name model_name ...>'
9     help = "Prints to console number of student related objects in a\
10 database."
11
12     models = (('student', Student), ('group', Group), ('user', User))
13
14     def handle(self, *args, **options):
15         for name, model in self.models:
16             if name in args:
```

```
17     self.stdout.write('Number of %ss in database: %d' %  
18         (name, model.objects.count())))
```

Знаходиться дана команда в модулі: “students/management/commands/stcount.py”.

Щоб протестувати код даної команди нам буде достатньо реалізувати один тестовий метод. Для цього давайте створимо новий модуль в пакеті із тестами. Назовемо його test_command.py, щоб було зрозуміло, що даний модуль містить тести команд.

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students/management/commands  
2  
3 $ touch test_command.py
```

Редагуємо новостворений модуль і набиваємо кодом. Але перед тим, як переглядати наступний кусок коду, попрошу вас спочатку самостійно спробувати написати даний тест. Витратьте хоча б годину власного часу і лише тоді продовжуйте читати далі...

Під час тестування команди маємо здійснити дві речі:

- запустити команду із переданими аргументами; для цього Django надає спеціальну функцію виклику команди з коду;
- перевірити результат роботи команди; у нашому випадку команда виводить інформацію у консоль; тому, щоб протестувати вивід маємо перекрити файл виводу даних власним.

Ось як виглядатиме тестовий метод у новому тестовому класі:

Тестуємо команду stcount

```
1 from django.core.management import call_command
2 from django.test import TestCase
3 from django.utils.six import StringIO
4
5
6 class STCountTest(TestCase):
7     """"Test stcount command"""
8
9     fixtures = [ 'students_test_data.json' ]
10
11     def test_command_output(self):
12         # prepare output file for command
13         out = StringIO()
14
15         # call our command
16         call_command('stcount', 'student', 'group', 'user', stdout=o\
17 ut)
18
19         # get command output
20         result = out.getvalue()
21
22         # check if we get proper number of objects in database
23         self.assertIn('students in database: 4', result)
24         self.assertIn('groups in database: 2', result)
25         self.assertIn('users in database: 1', result)
```

Давайте детальніше пройдемось по основних рядках коду:

- 1ий рядок: пакет django.core.management надає нам функцію `call_command`⁴³⁶; нею ми скористаємось для запуску команди `stcount`; дана функція надає увесь необхідний інструментарій роботи з Django командами;

⁴³⁶http://djbook.ru/rel1.7/ref/django-admin.html?#django.core.management.call_command

- Зій: імпортуємо файло-подібний клас `StringIO`⁴³⁷; з його допомогою створимо об'єкт в якості стрічкового буфера і записуватимемо дані виведені з команди `stcount`;
- бий: тестовий клас назовемо `STCountTest` і, як завжди, унаслідуємось від базового класу `TestCase`;
- 9ий: також встановлюємо атрибут класу `fixtures`, адже нам необхідні тестові дані для коректної роботи команди;
- 11ий: тестовий метод назовемо `test_command_output`, адже маємо єдину команду; в назві тестового методу можемо обійтися без назви команди, яку тестуємо;
- 13ий: створюємо файло-подібний об'єкт; ним ми перекриємо стандартний системний файл виводу, який виводить дані у консоль;
- 16ий: головна стрічка даного методу; тут ми запускаємо команду `stcount`; першим аргументом передаємо назву команди; усі наступні аргументи функції `call_command` будуть передані команді `stcount`; останній аргумент під назвою `stdout` дозволяє нам перекрити файл стандартного виводу власним файлом; тепер результат команди `stcount` попаде в змінну `out`, а не у консоль, де ви запустили тести;
- 20ий: згідно інтерфейсу об'єкта `StringIO` можемо скористатись методом `getvalue` і отримати текст, записаний в даний файл; зберігаємо результат в змінну `result`, щоб пізніше зробити перевірку;
- 23ій: перевіряємо чи команда `stcount` вивела правильну кількість студентів в базі;
- 24ий: подібним чином перевіряємо коректність команди на кількість груп та користувачів у базі даних.

Спробуйте тепер запустити новий тест і перевірте результат:

⁴³⁷<https://docs.python.org/2/library/stringio.html>

Запускаємо тест команди

```
1 python manage.py test students.tests.test_command
2 Creating test database for alias 'default'...
3 ...
4 INFO 2015-11-03 14:59:31,385 signals: Student added: Student1 Last N\
5 аme 1 (ID: 2)
6 ...
7 .
8 -----
9 --
10 Ran 1 test in 0.923s
11
12 OK
13 Destroying test database for alias 'default'...
```

Якщо все зроблено правильно, то отримаєте статус OK і успішно пройдений один тест.

Тепер ви знаєте як тестиувати команди. Час переходити до наступних компонент у нашому проекті.

Тест відправки листа

В даній секції ми, по-суті, будемо знову ж таки тестиувати в'юшку і форму. Форму контакту адміністратора. Чому ще раз тестиувати форму спитаєте ви? А тому, що дана форма відправляє лист. Тобто основний акцент даного тесту буде на перевірці відправки листа.

В Django фреймворку є спеціальна заглушка для відправки листів. В тестовому середовищі, під час прогонки тестів, вона активізується і жоден лист не вийде назовні. Натомість листи потрапляють у спеціальну змінну в пакеті `django.core.mail`. Це все відбувається завдяки тестовому емейл бекенду, що знаходиться в модулі `django.core.mail.backends.locmem`.

Отже, давайте розпочнемо новий тест. Знову ж таки, створюємо окремий тестовий модуль. Назовемо його `test_contact_admin.py`:

Створюємо файл `test_contact_admin.py` в пакеті `tests`

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students/tests
2 $ touch test_contact_admin.py
```

Відкриваємо модуль в редакторі і набиваємо заготовочкою:

Заготовка для тестового метода форми контакту адміністратора

```
1 from django.core import mail
2 from django.test import TestCase, Client
3 from django.core.urlresolvers import reverse
4
5 class ContactAdminFormTests(TestCase):
6
7     fixtures = ['students_test_data.json']
8
9     def test_email_sent(self):
10         """Check if email is being sent"""
```

Нічого нового там ми не знайдемо окрім імпорту `mail`. Задекларували клас тестового кейсу `ContactAdminFormTests` та тестовий метод `test_email_sent`. Тепер заповнимо тестовий метод необхідним кодом. Ось як повинна виглядати загальна картина:

- створюємо тестового клієнта, яким логуємось у сайт; адже пам'ятаємо, що контактувати адміністратора аноніми не можуть;
- постимо форму контакту із дійсними даними;
- перевіряємо чи список листів поповнився нашим листом.

А тепер і сам код тесту:

Тестуємо відправку листа

```
1 def test_email_sent(self):
2     """Check if email is being sent"""
3     # prepare client and login as administrator
4     client = Client()
5     client.login(username='admin', password='admin')
6
7     # make form submit
8     response = client.post(reverse('contact_admin'), {
9         'from_email': 'from@gmail.com',
10        'subject': 'test email',
11        'message': 'test email message'
12    })
13
14     # check if test email backend catched our email to admin
15     msg = mail.outbox[0].message()
16     self.assertEqual(msg.get('subject'), 'test email')
17     self.assertEqual(msg.get('From'), u'from@gmail.com')
18     self.assertEqual(msg.get_payload(), 'test email message',)
```

Тут також варто глянути детальніше на код:

- 4ий рядок: створюємо, так би мовити, тестовий браузер;
- 5ий: входимо у сайт під адміністратором, якого для нас створила тестова фікстура даних;
- 8ий: робимо запит типу POST на обробник форми контакту; тобто на в'юшку під назвою contact_admin; при цьому передаємо усі обов'язкові поля форми: адресу відправника, заголовок та тіло листа; метод post об'єкта клієнта ми уже з вами використовували під час тестування поста форми у попередніх секціях цієї глави; відповідь запам'ятали у змінну response; проте її не використовуватимемо далі в тестовому методі для перевірки; натомість перевірятимемо наявність листа в змінній пакету mail;

- 15ий: змінна `outbox`⁴³⁸, що міститься в пакеті `mail`, є масивом листів; оскільки ми лише один раз відправили листа, то і масив має складатись із одного об'єкта; даний об'єкт створений на базі класу `EmailMessage`⁴³⁹ і містить усю інформацію про лист; а його метод `message` поверне дану інформацію у зручному форматі;
- 16ий: витягуємо із повідомлення (zmінної `msg`) заголовок листа і перевіряємо чи співпадає із тим, що ми відправили на обробник форми контакту;
- 18ий: подібним чином перевіряємо тіло листа; щоб витягнути його з повідомлення, використовуємо метод `get_payload`⁴⁴⁰.

Запустіть новостворений тест і переконайтесь, що тест пройшов успішно.

Зауважте, що вміст змінної `outbox` очищується між кожним тестовим методом. Тому можемо гарантувати, що в нашому тестовому методі там знаходиться лише один лист - лист відправлений адміну.

Таким чином, ми зробили дію в одному місці системи (пост на в'юшку), а отримали результат в зовсім іншому (zmінній `outbox` в пакеті `django.core.mail`). Так. Іноді, щоб розібратись і писати якісні тести, приходиться краще володіти технологією, ніж при створенні самого функціоналу. Адже програміст, що пише юніт тести, зазвичай, повинен розуміти внутрішній код фреймворку і часто залазити туди, коли документації вже не вистачає. А таких випадків буває доволі багато.

Тест обробника сигналу

В нашій аплікації ми реалізували щонайменше дві функції-обробники сигналів. Одна відловлює редагування та додавання студентів, інша - видалення студентів. Обидві логують дану подію в консоль та у наш кастомний файл.

⁴³⁸ <http://djbook.ru/rel1.7/topics/testing/tools.html#email-services>

⁴³⁹ <http://djbook.ru/rel1.7/topics/email.html#django.core.mail.EmailMessage>

⁴⁴⁰ https://docs.python.org/2/library/email.message.html#email.message.Message.get_payload

Задачею даної секції буде протестувати функцію-обробник, яка слухає події додавання нового та редагування існуючого студентів. Давайте пригадаємо її код:

Обробник редагування та додавання студента

```
1 @receiver(post_save, sender=Student)
2 def log_student_updated_added_event(sender, **kwargs):
3     """Writes information about newly added or updated student into \
4     log file"""
5     logger = logging.getLogger(__name__)
6
7     student = kwargs['instance']
8     if kwargs['created']:
9         logger.info("Student added: %s %s (ID: %d)", student.first_n\
10 ame,
11                 student.last_name, student.id)
12 else:
13     logger.info("Student updated: %s %s (ID: %d)", student.first\
14 _name,
15                 student.last_name, student.id)
```

Функція `log_student_updated_added_event` слухає сигнал `post_save`, який охоплює як події редагування, так і події додавання нових об'єктів. Відповідно, в коді функції ми самостійно розпізнаємо створення нового об'єкта і логуємо його по-іншому.

Подумайте, як би ви тестиували подібну функцію? Данна функція, як і форма контакту адміністратора, здійснює певні дії, які впливають на інші аспекти середовища. У випадку з формою контакту лист зберігався у відповідну змінну на рівні пакету. У випадку із даним обробником сигналу результат його дії записується у двох місцях: файл та консоль.

Так ми налаштували логер для наших обробників сигналів:

Модуль settings.py проекту

```
1 LOG_FILE = os.path.join(BASE_DIR, 'studentsdb.log')
2
3 LOGGING = {
4     ...
5     'loggers': {
6         ...
7             'students.signals': {
8                 'handlers': ['console', 'file'],
9                 'level': 'INFO',
10            },
11            ...
12        }
13    ...
14 }
```

Таким чином, щоб перевірити роботу нашого обробника сигналу на коректність, потрібно або перекрити один із існуючих обробників логера ‘students.signals’, або додати ще один спеціально для тесту.

В даному випадку, найпростішим підходом буде самостійно зареєструвати в тестовому методі ще один обробник на кореневий логер. Що це нам дає? Такий підхід має одразу кілька переваг:

- кореневий логер ловить усі повідомлення в системі, а значить зловить і повідомлення від ‘students.signals’ логера;
- нам не потрібно буде перекривати існуючі логери чи хендлери;
- нам не потрібно буде редагувати LOGGING змінну в settings.py спеціально для потреб тестів.

Тепер до діла. Створіть самостійно новий модуль в корені підпакету tests і назвіть його test_signals.py. В ньому зберігатимемо тести на код кастомних обробників сигналів.

Знову ж таки, попрошу вас спочатку спробувати самостійно реалізувати даний тест. Попрацюйте над цим хоча б одну годину. І якщо нічого не вдається, лише тоді продовжуйте далі читати книгу. Такі вправи вже повинні вам бути під силу. Самостійні ідеї для тестування нових для вас компонент гарно розвивають вміння вирішувати поточні проблеми, що виходять за рамки документації та інструкцій.

Відкриваємо `test_signals.py` і розбираємо наступний код, що представляє повний код тесту функції `log_student_updated_added_event`:

Тестуємо обробник сигналу `log_student_updated_added_event`

```
1 import logging
2
3 from django.utils.six import StringIO
4 from django.test import TestCase
5
6 from students.models import Student
7 from students import signals
8
9
10 class StudentSignalsTests(TestCase):
11
12     def test_log_student_updated_added_event(self):
13         """Check logging signal for newly created student"""
14         # add own root handler to catch student signals output
15         out = StringIO()
16         handler = logging.StreamHandler(out)
17         logging.root.addHandler(handler)
18
19         # now create student, this should raise new message inside
20         # our logger output file
21         student = Student(first_name='Demo', last_name='Student')
22         student.save()
```

```
23
24     # check output file content
25     out.seek(0)
26     self.assertEqual(out.readlines()[-1],
27                       'Student added: Demo Student (ID: %d)\n' % student.id)
28
29     # now update existing student and check last line in out
30     student.ticket = '12345'
31     student.save()
32     out.seek(0)
33     self.assertEqual(out.readlines()[-1],
34                       'Student updated: Demo Student (ID: %d)\n' % student.id)
35
36     # remove our handler from root logger
37     logging.root.removeHandler(handler)
```

Перші рядки модуля ми пропустимо, адже у них немає нічого нового для нас. Стандартні імпорти та декларація тестового кейсу і тестового методу. А решту коду виконує наступну задачу:

- реєструємо наш власний хендлер на кореневий логер; даний хендлер записуватиме повідомлення у наш файловий об'єкт, який зможемо перевіряти без проблем;
- створюємо нового студента і записуємо в базу;
- перевіряємо чи наш файловий об'єкт містить повідомлення про нового студента;
- повторюємо процедуру, але тепер тестуємо редагування існуючого студента;
- не забуваємо видалити наш кастомний хендлер із кореневого логера.

Операції з логерами повинні бути вам уже знайомі з попередніх глав. Але все ж таки давайте глянемо на кілька основних рядків даного тесту:

- 15ий рядок: знову використаємо об'єкт класу `StringIO`; наш хендлер записуватиме повідомлення саме в даний файлоподібний об'єкт;

- 16ий: створюємо хендлер, який вміє писати повідомлення у потоки; в даному випадку це буде файловий об'єкт;
- 17ий: реєструємо новостворений хендлер на кореневий логер; таким чином, наш хендлер спрацьовуватиме на кожне повідомлення всередині фреймворку, адже кореневий логер перехоплює усі повідомлення в системі;
- 21ий: створюємо і зберігаємо нового студента, щоб спрацював наш обробник сигналу `post_save`;
- 25ий: перед тим, як перевіряти значення в змінній `out`, потрібно “перемотати” курсор файла на початок; інакше команда `readlines` може повернути нам порожню стрічку, навіть якщо файл і не є порожнім; поцікавтесь роботою з [файловими об'єктами в мові Python⁴⁴¹](#), якщо відчуваєте, що бракує теоретичних знань;
- 26ий: перевіряємо чи остання стрічка у файлі `out` відповідає повідомленню надісланому із обробника `log_student_updated_added_event` про створення нового студента; `readlines` повертає масив стрічок із файла і ми беремо останню стрічку з даного масиву;
- 30ий: подібним чином тестиємо чи отримаємо коректне повідомлення під час оновлення існуючого студента;
- 37ий: і не забуваємо забрати наш кастомний хендлер із кореневого логера.

Як бачите, ми змогли отримати повідомлення із обробника сигналу без додаткових змін в коді проекту. Все, що ми зробили - “навішали” додатковий хендлер на кореневий логер. А сам хендлер ми “попросили” записувати повідомлення у файлоподібний об'єкт, щоб пізніше мати змогу перевірити його вміст. Подібні цікаві рішення вам прийдеться знаходити щодня у вашій кар'єрі програміста.

Не забудьте запустити тест і переконатись, що він успішно проходить.

Домашнє завдання: аналогічним чином протестуйте решту обробників сигналів у проекті та коді аплікації `students`. Логіка і підхід у тестах будуть

⁴⁴¹https://www.ibm.com/developerworks/ru/library/l-python_part_8/

дуже схожими на те, що ми запрограмували протягом даної секції. Також після того, як виконаєте дане завдання, зможете звірити власний результат із кодом, що йде з книгою.

Тест процесора контексту

В Django аплікації `students` ми маємо один процесор контексту. Він передає змінну `GROUPS` в шаблони і називається `groups_processor`. Код процесора надзвичайно простий і тому його тест буде, мабуть, одним із найкоротших із тих, що ми писали в даній главі.

Давайте пригадаємо код процесора:

Код процесора контексту `groups_processor`

```
1 from .util import get_groups
2
3 def groups_processor(request):
4     return {'GROUPS': get_groups(request)}
```

Основна логіка процесора захована у функції `get_groups`, яка знаходиться в модулі `util`. А функції даного модуля ми уже покрили тестами. Тому основною нашою задачею буде поверхневий тест на перевірку вихідних даних функції `groups_processor` у потрібному форматі.

Знову ж таки, створюємо новий модуль для тестів: `test_context_processors.py` і набиваємо його новим класом і тестовим методом:

Тестуємо процесор контексту groups_processor

```
1 from django.test import TestCase
2 from django.http import HttpRequest
3
4 from students.context_processors import groups_processor
5
6
7 class ContextProcessorsTests(TestCase):
8
9     fixtures = [ 'students_test_data.json' ]
10
11     def test_groups_processor(self):
12         """Test groups processor"""
13         request = HttpRequest()
14         data = groups_processor(request)
15
16         # test data from processor
17         self.assertEqual(len(data['GROUPS']), 2)
18         self.assertEqual(data['GROUPS'][0]['title'], u'Group1')
19         self.assertEqual(data['GROUPS'][1]['title'], u'Group2')
```

Процесор контексту вимагає своїм єдиним аргументом об'єкт запиту (HttpRequest). Тому ми самостійно готуємо змінну request і передаємо під час виклику функції groups_processor.

Результат виклику даної функції ми перевіряємо згідно даних у базі. Наша тестова фікстура створила в базі дві групи: Group1 та Group2. Таким чином, перевіривши словник із групами ми можемо швидко переконатись у коректності роботи процесора.

Запускаємо даний тест і звіряємо результат:

```
1 $ python manage.py test students.tests.test_context_processors
2 Creating test database for alias 'default'...
3 ...
4 INFO 2015-11-04 11:49:28,314 signals: Student added: Student1 Last Name 1 (ID: 2)
5 ...
6 ...
7 .
8 -----
9 --
10 Ran 1 test in 0.261s
11
12 OK
13 Destroying test database for alias 'default'...
```

Тест кастомного фільтру

Остання компонента, яку ми покриємо тестами у даній главі книги буде кастомний шаблонний фільтр.

В аплікації `students` ми з вами реалізували дві кастомні компоненти: фільтр `str2int` та тег `pagenav`. Кастомний фільтр ми протестуємо в цій секції, а кастомний тег залишиться вам на домашнє завдання.

В коді, що йде із книгою можете знайти тести обидвох компонент, щоб звірити свій код на коректність.

Давайте пригадаємо, що собою представляє кастомний фільтр під назвою `str2int`:

Модуль templatetags/str2int.py

```
1 from django import template
2
3 register = template.Library()
4
5 @register.filter
6 def str2int(string):
7     """Convert input string into integer. If can not convert, return \
8     0"""
9     try:
10         value = int(string)
11     except ValueError:
12         value = 0
13     return value
```

Даний шаблонний фільтр пробує перетворити отриманий аргумент в ціле число. Якщо йому це не вдається, тоді він повертає нуль.

Тестувати фільтр можемо двома способами:

- просто викликати функцію фільтра із потрібним аргументом і перевірити чи отриманий результат буде коректним; тобто як звичайну Python функцію;
- проте можна скористатись даним фільтром в шаблоні і одним рухом протестувати не лише саму функцію фільтра, але й усю пов'язану із ним інфраструктуру: коректність реєстрації даної функції як фільтра у системі.

Другий спосіб не є юніт тестом, а швидше інтеграційним. Проте він дозволить протестувати усе одразу. Також звичайний тест функції ми з вами уже робили, тому буде корисно розібратись із тим, як на ходу з коду створювати куски Django шаблона без створення окремого файлу.

Почнемо знову із заготовки модуля:

Створюємо модуль для тесту кастомних фільтрів та тегів

```
1 $ cd /data/work/virtualenvs/studentsdb/src/studentsdb/students/tests
2 $ touch test_templatetags.py
```

Відкриваємо новий модуль і набиваємо тестом фільтра str2int:

Тестуємо str2int

```
1 from django.template import Template, Context
2 from django.test import TestCase
3 from django.core.paginator import Paginator
4
5
6 class TemplateTagTests(TestCase):
7
8     def test_str2int(self):
9         """Test str2int template filter"""
10        out = Template(
11            "{% load str2int %}"
12            "{% if 36 == '36'|str2int %}"
13            "it works"
14            "{% endif %}"
15        ).render(Context({}))
16
17        # check for our addition operation result
18        self.assertIn("it works", out)
```

Як бачите, ми реалізували доволі високорівневий тест, де пробуємо скористатись нашим фільтром всередині шаблону. Потім перевіряємо чи отримали на скомпільованому виході шаблона необхідний результат.

Давайте детальніше глянемо на новий для нас код:

- 10ий рядок: клас `Template`⁴⁴² представляє клас шаблону в Django; щоб відрендерити шаблон, потрібно пройти два етапи: 1. компілюємо код

⁴⁴²<http://djbook.ru/rel1.7/ref/templates/api.html#using-the-template-system>

шаблону (тобто створюємо об'єкт шаблона Template і передаємо йому сирий код на шаблонній мові; мікс HTML тегів та Django тегів), 2. рендеримо шаблон передаючи йому контекст (Context) із необхідними змінними;

- 11ий: в цій і наступних 4-ох стрічках ми передаємо код шаблону; в даному рядку завантажуємо наш кастомний тег str2int;
- 12ий: тут ми використовуємо фільтр str2int в умовному тезі; ми уже знаємо, що операція порівняння (=) в шаблоні поверне False, якщо операнди різних типів; у нашому випадку це стрічка ('36') і число (36);
- 13ий: якщо текст "it works" буде присутнім в кінцевому коді шаблона, значить наш фільтр str2int спрацював коректно;
- 15ий: після створення об'єкта Template маємо його відрендерити; метод render допоможе нам у цьому; рендер - це процедура інтерпретації шаблонної мови у кінцевий текст, при цьому враховуючи переданий контекст; в нашому випадку об'єкт контекст (Context) отримує порожній словник, адже наш шаблон не потребує жодних змінних;
- 18ий: наприкінці тестового методу проводимо перевірку; якщо текст "it works" знаходиться в кінцевій стрічці згенерованій шаблоном, значить наш фільтр str2int працює добре.

Тепер ви вмієте на ходу створювати шаблони та рендерити їх. Така процедура буває корисною, наприклад, якщо реалізуєте динамічний AJAX функціонал і відповідь із сервера необхідна в HTML форматі. Також у складних випадках може знадобитись особлива динаміка і генерація шаблонної логіки з Python коду. Але це вже поза межами даної книги.

Тепер самостійно запустіть тест. Не забувайте перед запуском тестів стартувати вашу базу даних. Переконайтесь, що тест пройшов успішно.

Нагадую, що тест кастомного тегу ваш залишається на домашнє опрацювання. В коді, що йде з книгою, зможете пізніше звірити свій результат.

...

На цьому ми завершуємо написання тестів. В наступній секції розглянемо як на практиці в Django проекті перевіряти рівень покриття коду тестами.

Покриття коду тестами

На початку глави ми коротенько обговорили, що таке Code Test Coverage (покриття коду тестами). В даній секції ми на практиці спробує підрахувати відсоток покритого коду тестами.

В Django та Python спільноті є вже цілий ряд своїх інструментів для розрахунку і зручного візуального відображення частини коду покритої тестами.

Базовим пакетом, що надає основний функціонал по перевірці покриття коду тестами є [coverage⁴⁴³](#). В Django спільноті створили ще один пакет, який є надбудовою над coverage і називається він [django-coverage⁴⁴⁴](#). Данна надбудова дозволяє, наприклад, через налаштування Django проекту (модуль settings.py) конфігурувати прогонку тестів при перевірці покриття.

Отже, почнемо із інсталяції необхідного софта. Для цього відкриваємо requirements.txt і додаємо обидва Python пакети наприкінці існуючого там списку пакетів:

requirements.txt

-
- 1 django-coverage==1.2.4
 - 2 coverage==3.7.1
-

Інсталюємо нові пакети:

Інсталюємо coverage пакети

-
- 1 \$ cd /data/work/virtualenvs/studentsdb/src/studentsdb
 - 2 \$ pip install -r requirements.txt
-

⁴⁴³<https://pypi.python.org/pypi/coverage>

⁴⁴⁴<https://pypi.python.org/pypi/django-coverage/>

Зверніть увагу, що потрібно використовувати версію пакета coverage із серії 3х. Версія 4 і вище не працюватимуть коректно в нашому розробницькому середовищі.

Це ще не все. Python пакети заінсталювані. Але один із них, а саме django-coverage, є також Django аплікацією. Саме тому маємо заінсталювати його також на рівні Django фреймворку. Надіюсь ви уже знаєте як це робити:

settings.py модуль в корені проекту studetsdb

```
1 INSTALLED_APPS = (
2     ...
3     'django_coverage',
4     'students',
5     'studentsdb',
6 )
```

Після останньої дії у вас з'явиться ще одна Django команда: test_coverage. Просто запустіть скрипт manage.py із опцією “help” і ви самі в цьому переконаєтесь:

Перевіряємо список доступних команд

```
1 $ python manage.py --help
2 ...
3 Available subcommands:
4 ...
5     validate
6
7 [django_coverage]
8     test_coverage
9
10 [registration]
11     cleanupregistration
12 ...
```

Можемо нею скористатись і оцінити результат:

Запускаємо тести в режимі перевірки покриття

```
1 $ python manage.py test_coverage students
```

Єдиним аргументом команді `test_coverage` ми передали назву аплікації для тесту. Адже ми не хочемо переглядати відсоток покриття по усьому Django проекту.

Приблизно наступний результат ви отримаєте після завершення команди `test_coverage`:

Результат команди `test_coverage`

1	Name	Stmts	Miss	Cover	Missing
2					
3	students.admin	20	20	0%	9-41
4	students.apps	4	4	0%	5-9
5	students.context_processors	2	0	100%	
6	students.management.commands.stcount	8	0	100%	
7	students.models.groups	7	7	0%	5-23
8	students.models.monthjournal	37	37	0%	5-53
9	students.models.students	12	12	0%	5-49
10	students.signals	10	2	80%	9, 22
11	students.templatetags.pagenav	15	2	87%	13-14
12	students.templatetags.str2int	7	2	71%	10-11
13	students.util	29	0	100%	
14	students.views.contact_admin	32	6	81%	67-71, \ 81-83
15					
16	students.views.groups	48	29	40%	18-30, \ 38-69, 77, 82-86
17					
18	students.views.journal	49	44	10%	20, 24-\ 106, 109-126
19					
20	students.views.students	88	45	49%	45-123, \ 164, 179, 184
21					
22					

```
23 TOTAL 368 210 43%
24
25 The following packages or modules were excluded: students.templatetags
26 __init__ students.management.commands.__init__ students.tests students.
27 management.__init__ students.views.__init__ students.models.__
28 init__ students.__init__ students.migrations students.locale
29
30 There were problems with the following packages or modules: students\
31 .fixtures students.templates students.static
```

Команда дала нам результат прямо в консоль у вигляді таблиці даних. В кожному рядочку маємо назву модуля, загальну кількість виразів (можна також сприймати це як кількість робочих рядків, тобто рядків з логікою; сюди не входять імпорти, коментарі і інші рядки коду, що не містять бізнес логіки), кількість виразів не покрита тестами, відсоток виразів покритих тестами та конкретні рядки, що не були покриті.

По даному звіту можете бачити, де слід допрацювати по ваших тестах.

Але, знову ж таки, хочу наголосити, що покриття коду тестами не є однозначним показником якості коду. Тому покриття в районі 50-70% зазвичай є більш, ніж достатнім у реальних проектах.

Як бачите, даний формат звіту про покриття коду тестами є не надто зручним і наглядним. І пакети coverage та django-coverage уже дають нам інструменти для кращого візуального представлення цього звіту.

Одним із них є дуже зручний формат HTML. Команда test_coverage може згенерувати для нас набір HTML файлів, які зможемо переглянути у веб-переглядачі. Для цього необхідно лише додати один параметр в налаштуваннях проекту:

Налаштовуємо шлях до папки з HTML звітом, settings.py

```
1 COVERAGE_REPORT_HTML_OUTPUT_DIR = os.path.join(BASE_DIR, '.', 'cove\\'
2 rage')
```

Змінна COVERAGE_REPORT_HTML_OUTPUT_DIR вказує команді `test_coverage` згенерувати HTML звіт замість текстового звіту в командній стрічці. Також з допомогою даної змінної ми визначаємо шлях до папки, в якій зберігатимуться згенеровані файли.

Як бачите, ми вказали папку `coverage` поза репозиторіем коду. Уесь звіт про покриття коду є генерованим, тому немає змісту тримати такі файли у репозиторії.

Після останніх змін ще раз запустіть команду `test_coverage`. І цього разу замість звіту в консолі отримаєте наступне повідомлення:

Запускаємо `test_coverage`

```
1 ...
2 .
3 -----
4 --
5 Ran 19 tests in 3.764s
6
7 OK
8 Destroying test database for alias 'default'...
9
10 HTML reports were output to '/data/work/virtualenvs/studentsdb/src/c\\
11 overage'
```

В папці “`src/coverage`” отримаєте набір файлів. Серед них буде файл під назвою `index.html`. Саме його потрібно відкрити у своєму браузері. На вас чекатиме подібна картинка:

Test Coverage Report
Generated: Thu 2015-11-05 15:11 UTC
42% coverage

Module	Statements			% covered
	total	executed	excluded	
students.admin	25	0	5	0.0%
students.apps	7	0	3	0.0%
students.context_processors	3	2	1	100.0%
students.management.commands.stcount	11	8	3	100.0%
students.models.groups	14	0	7	0.0%
students.models.monthjournal	41	0	4	0.0%
students.models.students	16	0	4	0.0%
students.signals	14	8	4	80.0%
students.templatetags.pagenav	16	13	1	86.7%
students.templatetags.str2int	8	5	1	71.4%
students.util	33	29	4	100.0%
students.views.contact_admin	43	26	11	81.2%
students.views.groups	60	19	12	39.6%

Покриття тестами аплікації students, HTML формат

На даній сторінці можете бачити загальний процент покриття аплікації students тестами, а також список модулів і дані по кожному із них. Клацнувши на будь-який із модулів отримаєте деталізацію покриття коду по даному модулю:

```
students.views.contact_admin << index >> students.views.journal

students.views.groups: 60 total statements, 39.6% covered
Generated: Thu 2015-11-05 15:11 UTC
Source file: /data/work/virtualenvs/studentsdb/src/studentsdb/students/views/groups.py
Stats: 19 executed, 29 missed, 12 excluded, 41 ignored

1. from django.shortcuts import render
2. from django.http import HttpResponseRedirect, HttpResponseRedirect
3. from django.core.urlresolvers import reverse
4. from django.core.paginator import Paginator, EmptyPage, PageNotAnInteger
5. from django.forms import ModelForm
6. from django.views.generic import CreateView, UpdateView, DeleteView
7. from django.utils.translation import gettext as _
8.
9. from crispy_forms.helper import FormHelper
10. from crispy_forms.layout import Submit
11. from crispy_forms.bootstrap import FormActions
12.
13. from ..models import Group
14. from ..util import paginate
15.
16.
17. def groups_list(request):
18.     groups = Group.objects.all()
19.
20.     # try to order groups list
21.     order_by = request.GET.get('order_by', '')
22.     if order_by in ('title',):
```

Сторінка з деталями по окремому модулю

Погодьтесь, що даний формат є значно простішим і приємнішим для сприйняття.

Таким чином, після оновлення коду проекту, чи додавання нових тестів на існуючий коду, розробник запускає команду `test_coverage` і перевіряє поточний стан. Такий звіт дозволяє легко виявляти прогалини, якщо критичний код проекту не є покритий тестами.

На домашнє завдання також згенеруйте звіт на покриття тестами коду проекту `studentsdb`.

Домашнє завдання

Ми з вами дуже продуктивно провели час розбираючи тести! Крім того, що розібралися із загальним поняттям тестів і їхнім призначенням, змогли протестувати більшість існуючих компонент у нашому Django проекті:

- звичайні Python функції та класи;
- в'юшки та форми;
- моделі;
- кастомний фільтр;
- відправку емейлів;
- код обробника сигналу;
- Django команду;
- та процесор контексту.

Після такого курсу те, що залишилось протестувати буде для вас доволі простим завданням. Головне, що ми розібрали основний підхід та попрактикувались у роботі із тестовим середовищем.

...

Підсумуємо список речей на домашнє завдання.

Окрім того, що ви зустрічали протягом даної глави, попрошу самостійно попрацювати і:

- спробувати по максимуму покрити тестами код аплікації `students`;
- спробувати по максимуму покрити тестами код проекту `studentsdb`;
- почитати, самостійно розібратись і налаштувати на своєму комп’ютері сервіс для автоматичного запуску тестів проекту при кожному коміті у репозиторії; можете обрати для цього [Jenkins⁴⁴⁵](#), або будь-який інший існуючий інструмент;
- самостійно розібратись із тестами локалізації в Django і протестувати наші переклади на українську та англійську мови.

100%-ве покриття коду тестами немає особливого змісту в реальних проектах і не повертає відповідних дивідендів. Прошу вас спробувати покрити весь наш код тестами виключно з метою практики та набивки руки. Не ігноруйте дане завдання. Подібна рутина дає надзвичайно хороші результати при навчанні.

...

Наступна глава буде останньою технічною главою і в ній ми розберемо базовий процес переносу і запуску робочого проекту на кінцевий сервер. При цьому ми розглянемо кілька різних варіантів запуску продакшин сайту. Буде цікаво і пізнавально.

Але перед тим, як переходити далі, працюємо над домашками даної глави і стаємо профіками в автоматичному тестуванні!

⁴⁴⁵<https://jenkins-ci.org/>