Ajax+js+django = удобная админка

Продолжаем работу над проектом. Делаем админку для работы с зарегистрированными пользователями. Реализуем работу с пользователями без ajax и через ajax. Несколько слов о javascript.

[Форма регистрации](#_c713xnuw3xem)

[Изобретаем велосипед](#_xcxo7ce7t9im)

[Django-Forms](#_2sbjv4vh726)

[Работа с формами (Теория)](#_rqu8awfl30wv)

[Работа с формами (практика)](#_bufzbbdn94xb)

[Собственная админка](#_8dc50fmlstv0)

[Часть первая](#_k60dp4xxdala)

[Часть вторая(с ajax)](#_6via0e293kz1)

[Тег include](#_m8czxyqg7i9j)

[Отправка/прием JSON](#_cdyxmeeoe7l)

[Некоторые хитрости](#_hb5npga9tfbp)

[Редактирование и создание - практически одно и то же](#_jbs3jqs24vyh)

[Соберем все вместе](#_9yoxedwcggik)

[Домашнее задание](#_owpsqd9opgcf)

[Дополнительные материалы](#_y937sk8fclye)

[Используемая литература](#_fyo8hnwoo1bu)

# Форма регистрации

На предыдущем уроке мы сделали авторизацию пользователей. Теперь нужно дать возможность регистрироваться новым посетителям.

## Изобретаем велосипед

Чтобы лучше понять прицип работы Django, создадим и обработаем форму вручную.

Создадим html-страницу с формой регистрации registration\_low.html

|  |
| --- |
| {% extends "base.html" %}  {% block content %}  <form method="post" role="form" action="/user/registration/">  {% csrf\_token %}  <p>  <input type="text" name="name" placeholder="Enter Name" required>  </p>  <p>  <input type="email" name="email" placeholder="Enter Email" required>  </p>  <p>  <input type="email" name="confirm\_email" placeholder="Confirm Email" required>  </p>  <p>  <input type="password" name="password" placeholder="Enter password" required>  </p>  <p>  <input type="password" name="confirm\_password" placeholder="Enter password again" required>  </p>  <button type="submit" name="submit">Send</button>  </form>  {% endblock %} |

Самая простая форма с набором полей.

Добавим функцию-обработчик, принимающую данные из формы, и создающую пользователя в базе  
добавим во views.py

|  |
| --- |
| def registration\_low(request):  if request.method == 'POST':  errors = {} # Тут будем хранить ошибки, чтобы отобразить на странице  username = request.POST.get('name')  email = request.POST.get('email')  email2 = request.POST.get('confirm\_email')  password = request.POST.get('password')  password2 = request.POST.get('confirm\_password')  print(request.POST)  # Validate data  if email != email2:  errors['email'] = 'does not match'  if password != password2:  errors['password'] = 'does not match'  user = User(username=username, email=email)  # Пароли хранятся в виде хэшей, поэтому их нельзя передавать напрямую  user.set\_password(password)  # Проверяем, существует ли пользователь с таким именем  try:  user.validate\_unique()  except ValidationError as er:  errors.update(er.message\_dict)  # Если есть ошибки, передаем их в контексте шаблону, который умеет их отображать  if errors:  return render(request, 'registration\_low.html', {'reg\_errors': errors})  # Если ошибок нет, сохраняем пользователя в базе, перенаправляем на главную  user.save()  return HttpResponseRedirect("/")  return render(request, 'registration\_low.html') |

Это обычный python-код, в нем нет ничего сложного…  
**Важно:** Привыкайте не доверять полученным от пользователя данным , пока они не будут проверены!

Наша функция, прежде чем создать пользователя в базе, проверяет правильность введенных данных, а именно:

* совпадение emal’ов;
* совпадение паролей;
* уникальность имени пользователя.

Дополним шаблон, чтобы он умел отображать ошибки, добавим registration\_low.html

|  |
| --- |
| {% extends "base.html" %}  {% block content %}  {% if reg\_errors %}  <ul>  {% for key, value in reg\_errors.items %}  <li>{{ key }} - {{ value }}</li>  {% endfor %}  </ul>  {% endif %}  <form method="post" role="form" action="/user/registration/">  ... |

Код дан в сокращении

Это, конечно, не очень красиво, не совсем доделано и удобно для пользователя, но вполне работает! Можно продолжить дополнять шаблоны, дописывать проверку данных (например, у нас на стороне сервера не проверяется корректность email) и, приложив еще некоторое количество усилий, вы получите вполне симпатичную и полностью рабочую форму.

Но django предоставляет более удобные инструменты для подобной задачи django-forms. Их мы рассмотрим ниже, а пока обобщим принцип обработки данных из формы:

1. Мы предоставляем пользователю форму для ввода.
2. Данные из формы передаются в соответствующую функцию-обработчик.
3. Данные валидируются (проверяются на правильность, в соответствии с заданными критериями).
4. Исползуя Модель (мы использовали готовую модель User), создаём python-объект и, вызвав методы .save() , сохраняем данные в БД.

## Django-Forms

Выполним ту же задачу, но используя формы.

### Работа с формами (Теория)

Работа с формами – довольно не простая задача.. Необходимо подготовить для отображения в форме большое количество данных разного типа, отобразить форму в HTML, создать удобный интерфейс для редактирования, получить данные на сервере, проверить и преобразовать в нужный формат, и в конце сохранить или передать для дальнейшей обработки.

Формы Django могут упростить и автоматизировать большую часть этого процесса, и могут сделать это проще и надёжнее, чем код, написанный большинством программистов.

Django позволяет:

* подготовить данные для отображения в форме;
* создать HTML формы для данных;
* получить и обработать отправленные формой данные.

Вы можете написать код, который всё это будет делать, но Django может выполнить большую часть самостоятельно.

Сердце всего механизма – класс Form. Как и модель в Django, которая описывает структуру объекта, его поведение и представление, Form описывает форму, как она работает и показывается пользователю.

Как поля модели представляют поля в базе данных, поля формы представляют HTML <input> элементы. (ModelForm отображает поля модели в виде HTML <input> элементов, используя Form. Используется в админке Django.)

Поля формы являются классами. Они управляют данными формы и выполняют их проверку при отправке формы. Например, DateField и FileField работают с разными данными и выполняют разные действия с ними.

Поле формы представлено в браузере HTML “виджетом” - компонент интерфейса. Каждый тип поля представлен по умолчанию определенным классом Widget, который можно переопределить при необходимости. Т.е. поле формы умеет отображать себя в виде определенного html-кода.

### Работа с формами (практика)

Все формы принято хранить в файле forms.py, который размещается в соответствующем приложении (App). Для работы со встроенной моделью User, есть готовая форма UserCreationForm, но мы её немного модифицируем, чтобы продемонстрировать гибкость форм.

forms.py

|  |
| --- |
| from django import forms  from django.contrib.auth.models import User  from django.contrib.auth.forms import UserCreationForm  class MyRegistrationForm(UserCreationForm):  email = forms.EmailField(required=True)  first\_name = forms.CharField(required=False)  last\_name = forms.CharField(required=False)  class Meta:  model = User  fields = ('username', 'email', 'password1', 'password2')  def save(self, commit=True):  user = super(MyRegistrationForm, self).save(commit=False)  user.email = self.cleaned\_data['email']  user.first\_name = self.cleaned\_data['first\_name']  user.last\_name = self.cleaned\_data['last\_name']  if commit:  user.save()  return user |

Мы добавили возможность заполнения полей “Имя” и “Фамилия”(сделали эти поля необязательными required=False) и поле email (сделав его обязательным). Чтобы данные из добавленных полей сохранялись в базе, расширили метод save().

В классе Meta указываем модель, которую используем. А в атрибуте fields список полей, отображаемых в htm-форме (виджеты этих полей будут выбраны на основании типа полей).

Создаем шаблон для отображения формы registration.html

|  |
| --- |
| {% extends "base.html" %}  {% block content %}  <form method="post" role="form" action="/user/registration/">  {% csrf\_token %}  <table>  {% for field in form %}  <tr>  <td>  {{ field.label\_tag }}  </td>  <td>  <div>  <ul>  {% for error in field.errors %}  <li class="error">errors:{{ error }}</li>  {% endfor %}  {{ field }}  </ul>  </div>  </td>  </tr>  {% endfor %}  </table>  <input type="submit" value="Create the account">  </form>  {% endblock %} |

{{ field.label\_tag }} - Отображает метку поля формы  
{{ field }} - отображает виджет поля формы

{% for error in field.errors %}  
 <li class="error">errors:{{ error }}</li>  
{% endfor %} - отображает список всех ошибок для данного поля формы

Функция-обработчик выглядит так:

|  |
| --- |
| from MainApp.forms import MyRegistrationForm  def registration(request):  if request.method == 'POST':  form = MyRegistrationForm(request.POST)  if form.is\_valid():  form.save()  return HttpResponseRedirect('/')  context = {'form': form}  return render(request, 'registration.html', context)  context = {'form': MyRegistrationForm()}  return render(request, 'registration.html', context) |

form.is\_valid() - возвращает True, если валидация (проверка) данных прошла без ошибок.  
form.save() - сохраняет объект в БД, используя указанную, при создании формы, модель

Ошибки валидации хранятся в атрибуте .errors каждого поля формы.

Работа с формами, используя django-forms, позволит вам избежать множества рутинных операций, писать более грамотный и расширяемый код. Это очень мощный инструмент!

Подробнее о работе с формами и создании форм из моделей читайте: [здесь](http://djbook.ru/rel1.7/topics/forms/) и [здесь](http://djbook.ru/rel1.6/topics/forms/modelforms.html).

# Собственная админка

## Часть первая

Пользователи могут свободно регистрироваться и авторизовываться на нашем сайте. Но у нас нет инструмента для просмотра и редактирования зарегистрированных пользователей, мы даже не можем забанить того, кто плохо себя ведет.

Встраивать админку в общий интерфейс сайта не очень целесообразно, админка - это инструмент администраторов сайта, здесь лучше разработать отдельный дизайн. Админка - это инструмент, который непрерывно расширяется и дорабатывается.

Файл base\_admin.py я заполню очень примитивным дизайном, будет время - доработаю (Помните? Для нас главное - функционал.).

Шаблон админки admin\_page.py

|  |
| --- |
| {% extends "base\_admin.html" %}  {% block content %}  <div class="row">  <div class="col-md-6">  Пользователи:  <ul id="users">  {% for user in users %}  <li>{{ user }}  <button type="button" class="btn btn-default" aria-label="Edit">  <span class="glyphicon glyphicon-credit-card" aria-hidden="true"></span>  </button>  <a href="delete/user/{{ user.id }}" class="btn btn-default">  <span class="glyphicon glyphicon-remove" aria-hidden="true"></span>  </a>  </li>  {% endfor %}  </ul>  </div>  </div>  {% endblock %} |

<span class="...” - это просто иконки из bootstrap, если вы решили не пользоваться бутстрапом, просто вставьте подходящие иконки.

Админка будет отображать список всех пользователей с парой иконок для удаления и редактирования пользователя.

Обратите внимание на адрес запроса удаления пользователя: delete/user/{{ user.id }} . Если вы смотрели дополнительные материалы по обработке url (я надеюсь, что вы их смотрели, если нет, то самое время [посмотреть](http://djbook.ru/rel1.7/topics/http/urls.html#named-groups)), то вы знаете, что для обработки подобных url’ов можно использовать группы.

дополлняем urls.py

|  |
| --- |
| url(r'^admin/$', admin\_page),  url(r'^admin/delete/user/(\d+)$', delete\_user), |

views.py

|  |
| --- |
| def admin\_page(request):  # TODO: сделать доступ у админке только суперпользователю  users = User.objects.all()  return render(request, 'admin\_page.html', {'users': users})  def delete\_user(request, user\_id):  user = get\_object\_or\_404(User, id=user\_id)  user.delete()  return HttpResponseRedirect('/admin') |

Имя функции get\_object\_or\_404() говорит само за себя

Первая часть готова, теперь можно просматривать и удалять пользователей.

Выглядит просто, но здесь есть существенный недостаток. При любой операции мы отправляем запрос, а django полностью пересобирает страницу и отправляет назад, т.е., при “любом чихе” страница полностью пересоздается. Это подходит для простых и небольших админок, но, если админка довольно сложна и содержит большое количество информации, такой подход становится крайне не эффективен.

Зачем обрабатывать всю информацию, если можно изменить только то, что поменялось? В этом случае нам поможет технология ajax.

## Часть вторая(с ajax)

[Ajax](https://ru.wikipedia.org/wiki/AJAX) - подход к построению интерактивных пользовательских интерфейсов веб-приложений, заключающийся в «фоновом» обмене данными браузера с веб-сервером. В результате при обновлении данных веб-страница не перезагружается полностью, и веб-приложения становятся быстрее и удобнее.

Для работы в Ajax нужно использовать javascript. Если вы его не знаете или знаете очень плохо - ничего страшного, ничего сложного мы на нём писать не будем, и по комментариям вы сможете разобраться в коде. В любом случае, если вы собираетесь работать в мире web, от javascript’а вам никуда не деться, хотите вы того или нет.

Суть похода, которым будем пользоваться для создания и редактирования пользователей в админке, следующая: с django-сервером наш html-клиент будет обмениваться только информацией, в формате JSON, и изменять Html-страницу с помощью JS (javascript), модифицирую DOM-дерево. JSON - очень простой формат, о нём можно почитать [здесь](http://www.json.org/json-ru.html). Что такое DOM-дерево? Это структура html-документа, изменяя элементы этого дерева, можно динамически изменять Html-тсраницу. На чистом JS довольно неудобно работать с DOM-деревом, поэтому мы будем пользоваться библиотекой [JQuery](https://jquery.com/).

### Тег include

Нам понадобится использовать одни и те же блоки html кода в различных местах. Мы, конечно, можем их копировать, но чем грозит дублирование кода, надеюсь, вы знаете.

Чтобы избежать дублирования, воспользуемся шаблонным тегом include. Работает он очень просто: мы выносим нужный нам кусок html-кода в отдельный файл, а в том месте, куда нужно вставить, просто пишем {% include '<имя файла>.html' %} и django, при сборе шаблона, встроит указанный кусок. Этот тег используют не только для борьбы с дубликатами, но и для разбивки проекта на логические куски, для простоты поиска и модификации .

### Отправка/прием JSON

На стороне django получение и отправку JSON-данных демонстрирует код:

|  |
| --- |
| def send\_json(request):  # Если данные были отправлены ajax'ом  if request.is\_ajax():  # Данные хранятся также в атрибуте POST или GET, в зависимости от методы отправки данных  request\_data = request.POST  # Словарик при отправке автоматически будет преобразован к json  send\_data = {'key': 'value'}  return JsonResponse(send\_data)  raise Http404 |

На стороне клиента демонстрируется такой код:

|  |
| --- |
| function demo\_send\_data(id){  $.ajax({  url: '<адрес ajax-запроса>',  type: 'GET/POST',  dataType: 'json',  // Функция, которая будет выполнена при успешном ответе  success: function (response) {  console.log("response = ", response);  },  // Функция, которая будет выполнена при ошибке на сервере  error: function (xhr, status, error) {  console.log('error =', error)  }  });  } |

Пользуясь предоставленными примерами, как заготовками, можно легко обмениваться данными.

### 

### 

### Некоторые хитрости

Для упрощения JS-кода мы применим один хитрый трюк: при изменении данных о пользователях мы будем “перерисовывать” список пользователей, используя шаблонизатор django, но будем обновлять не всю страницу целиком, а только часть.

Для этого вынесем шаблон, отображающий список пользователей в отдельный html-файл, и встроим этот код в нужные места, а когда нужно будет обновить список на странице админки, попросим django собрать html для этого куска

inc-registration\_form.html (файлы для встраивания, я называю с приставки inc)

|  |
| --- |
| <div class="row">  <div class="col-md-6">  Пользователи:  <ul id="users">  {% for user in users %}  <li>{{ user }}  <button type="button" class="btn btn-default" aria-label="Edit"  onclick="fill\_form({{ user.id }})">  <span class="glyphicon glyphicon-credit-card" aria-hidden="true"></span>  </button>  <a href="delete/user/{{ user.id }}" class="btn btn-default">  <span class="glyphicon glyphicon-remove" aria-hidden="true"></span>  </a>  </li>  {% endfor %}  </ul>  </div>  </div> |

Теперь registration.html выглядит так

|  |
| --- |
| {% extends "base.html" %}  {% block content %}  {% include 'inc-registration\_form.html' %}  {% endblock %} |

Когда нужно будет получить список пользователей, воспользуемся следующим кодом

|  |
| --- |
| users = User.objects.all()  html = loader.render\_to\_string('users\_list.html', {'users': users}, request=request)  data = {'errors': False, 'html': html}  return JsonResponse(data) |

### Редактирование и создание - практически одно и то же

Мы уже знаем, как можно создавать объекты. Редактирование - практически то же создание. Для редактирования мы получаем объект из БД, меняем часть его полей и сохраняем назад. На практике, редактированием и созданием объекта занимается одна и таже функция-обработчик. У функции есть необязательный атрибут id, если id получен, то мы редактируем объет с указанным id, если нет - создаем новый.

### Соберем все вместе

Собрав вместе выше изложенное, получаем admin\_page.html

|  |
| --- |
| {% extends "base\_admin.html" %}  {% block js %}  {{ block.super }}  <script src="/static/js/create\_user.js"></script>  {% endblock %}  {% block content %}  <div class="row">  <div class="col-md-6">  Пользователи:  <ul id="users">  {% include 'inc-users\_list.html' %}  </ul>  </div>  </div>  <form id="user\_form" onclick="send\_data({{ id }})">  {% include 'inc-registration\_form.html' %}  <input type="submit" value="Edit account">  </form>  {% endblock %} |

дополняем views.py

|  |
| --- |
| def create\_user(request, user\_id=None):  """  Создает Пользователя(User)  Или редактирует существующего, если указан user\_id  """  if request.is\_ajax():  if not user\_id:  user = User(request.POST)  else:  user = get\_object\_or\_404(User, id=user\_id)  user = MyRegistrationForm(request.POST or None, instance=user)  if user.is\_valid():  user.save()  users = User.objects.all()  html = loader.render\_to\_string('users\_list.html', {'users': users}, request=request)  data = {'errors': False, 'html': html}  return JsonResponse(data)  else:  errors = user.errors.as\_json()  return JsonResponse({'errors': errors})  raise Http404  def get\_user\_form(request, user\_id):  """  Возвращает заполненную форму для редактирования Пользователя(User) с заданным user\_id  """  if request.is\_ajax():  user = get\_object\_or\_404(User, id=user\_id)  user\_form = MyRegistrationForm(instance=user)  context = {'form': user\_form, 'id': user\_id}  context.update(csrf(request))  html = loader.render\_to\_string('inc-registration\_form.html', context)  data = {'errors': False, 'html': html}  return JsonResponse(data)  raise Http404 |

edit\_user.js

|  |
| --- |
| function fill\_form(id){  $.ajax({  url: 'get\_user\_form/' + id,  type: 'GET',  dataType: 'json',  success: function (response) {  //console.log("response = ", response);  if (response.errors) {  console.log("errors = ", errors);  } else {  $('#user\_form').html(response.html);  }  },  error: function (xhr, status, error) {  console.log('error =', error)  }  });  } |

Полный пример мы можете скачать из приложения “исходные коды”, запустить и проверить.

# Домашнее задание

Продолжаем разработку своего проекта:

1. Добавить на свой сайт форму регистрации.
2. Пользуясь примерами данного урока, начать разработку собственной админки. Админка должна позволять:
   1. Просматривать всех зарегистрированных пользователей.
   2. Удалять пользователей.
   3. Редактировать параметры пользователя.
   4. \*(усложненное) отображать ошибки редактирования пользователя, если данные отправлены не корректно.

Все проблемные или не до конца понятные элементы снабжайте комментариями. Что-то хотели сделать, но не получилось, не удаляйте код - закомментируйте, снабдив подробными комментариями. При проверке ДЗ я постараюсь решить ваши проблемы и рассказать о наиболее частых проблемах в начале следующего занятия.

# Дополнительные материалы

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

1. [Работа с формами](http://djbook.ru/rel1.7/topics/forms/)
2. [Создание форм из моделей](http://djbook.ru/rel1.6/topics/forms/modelforms.html)
3. [JSON](http://www.json.org/json-ru.html)

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [Django Book(rus)](http://djbook.ru/)
2. [Django + Json(stackoverflow)](http://stackoverflow.com/questions/9262278/django-view-returning-json-without-using-template)
3. [Django Forms](http://djbook.ru/ch07.html)