Rozwiązania szkieletowe w tworzeniu aplikacji www

Tomasz Łukaszuk

Katedra Oprogramowania
Wydział Informatyki
Politechnika Białostocka

Temat wykładu:

Formularze

Fundusze Europejskie dla Rozwoju Społecznego
Rzeczpospolita Polska
Dofinansowane przez Unię Europejską
NCBR Narodowe Centrum Badań i Rozwoju

Materiały przygotowane w ramach projektu "PB 5.0 – dostosowanie oferty dydaktycznej Politechniki Białostockiej do potrzeb nowoczesnej gospodarki oraz zielonej i cyfrowej transformacji" (nr umowy FERS.01.05-IP.08-0327/23-00) w ramach programu Fundusze Europejskie dla Rozwoju Społecznego 2021-2027 współfinansowanego ze środków Europejskiego Funduszu Społecznego Plus.

Formularze: Agenda

1. Formularze w Django
   - Wprowadzenie
   - Formularze dla modeli
   - Form Media
   - Formsets
   - Przesyłanie plików
2. Formularze w RoR
   - Podstawy
   - Formularze powiązane z modelami
   - Pola typu select
   - Helpery do obsługi pól data i czas
   - Przesyłanie plików
   - Wyświetlanie błędów walidacji

Formularze w Django

Wprowadzenie

Formularze w HTML'u

```
1  <form action="/admin/" method="post">
2          <input id="your_name" type="text" name="your_name">
3          <input id="your_password" type="password" name="your_password">
4          <input type="submit" value="Log in">
5  </form>
```

Formularz - zbiór elementów, które umożliwiają odwiedzającemu wykonywanie takich czynności jak wprowadzanie tekstu, wybieranie opcji, manipulowanie obiektami lub kontrolkami, itp., a następnie wysyłanie tych informacji do serwera

`django.forms` - biblioteka do obsługi formularzy

`django.forms` pozwala na:

1. Wyświetlanie kodu HTML formularza z automatycznie generowanymi kontrolkami.
2. Sprawdzanie danych wysyłanych z formularza pod kątem spełniania określonych reguł walidacyjnych.
3. Ponowne wyświetlenie formularza w przypadku błędu walidacji.
4. Konwersję danych z formularza do odpowiednich typów danych Pythona.

4 podstawowe koncepcje

`django.forms`:

1. **Widget**
   Klasa, która odpowiada za kontrolkę formularza, np. `<input type="text">` lub `<textarea>`. Zajmuje się renderowaniem kontrolki do kodu HTML.
2. **Field**
   Klasa, która jest odpowiedzialna za walidację, np. EmailField zapewnia, że jego dane to prawidłowy adres e-mail.
3. **Form**
   Zbiór pól, który wie jak mają być one walidowane i wyświetlana jako HTML.
4. **Form Media**
   Zasoby CSS i JavaScript wymagane do renderowania formularza.

Obiekty Form

Obiekt Form zawiera pewną sekwencję pól formularza i zbiór reguł do sprawdzania poprawności. Reguły muszą zostać spełnione, aby formularz został zaakceptowany.

```
1  from django import forms
2
3  class ContactForm(forms.Form):
4      subject = forms.CharField(max_length=100)
5      message = forms.CharField()
6      sender = forms.EmailField()
7      cc_myself = forms.BooleanField(required=False)
```

Użycie formularza w funkcji widoku

```
1  def contact(request):
2      if request.method == 'POST':
3          form = ContactForm(request.POST)
4          if form.is_valid():
5              # Process the data in form.cleaned_data
6              # ...
7              return HttpResponseRedirect('/thanks/')
8      else:
9          form = ContactForm()
10
11     return render(request, 'contact.html', {'form': form,})
```

Trzy ścieżki kodu:

1. Utworzenie niepowiązanej (unbound) instancji ContactForm (GET) i przekazanie jej do templatki
2. Utworzenie powiązanej (bound) instancji ContactForm (POST), walidacja jej, przekierowanie do strony z podziękowaniem po pozytywnej walidacji
3. Jeśli formularz jest niepoprawny, przekazanie powiązanej instancji do templatki

Przetwarzanie danych z formularza

```python
from django.core.mail import send_mail

if form.is_valid():
    subject = form.cleaned_data['subject']
    message = form.cleaned_data['message']
    sender = form.cleaned_data['sender']
    cc_myself = form.cleaned_data['cc_myself']

    recipients = ['info@example.com']
    if cc_myself:
        recipients.append(sender)

    send_mail(subject, message, sender, recipients)
    return HttpResponseRedirect('/thanks/') # Redirect after POST
```

`is_valid()` nie tylko weryfikuje poprawność danych, ale również konwertuje je do odpowiednich typów Pythona.

Trzy typy walidacji

1. Metoda `clean()` w podklasie Field. Zwraca przetworzone (clean) dane.
2. Metoda `clean_<fieldname>()` w podklasie Form. Musi posłużyć się `self.cleaned_data` (zawiera obiekty Pythona). Zwraca przetworzone (clean) dane.
3. Metoda `clean()` w podklasie Form. Może wykonywać walidację na wielu polach na raz. Zwraca przetworzoną listę `clean_data`.

Wszystkie powyższe metody mogą generować wyjątki `ValidationError`, które zostaną dodane jako elementy odpowiedniej listy błędów (errorlist).

Wyświetlanie formularza z użyciem szablonu

Użycie `as_p`:

```
<form action="/contact/" method="post">
{{ form.as_p }}
<input type="submit" value="Submit" />
</form>
```

odpowieadający kod HTML:

```
<form action="/contact/" method="post">
<p><label for="id_subject">Subject:</label>
    <input id="id_subject" type="text" name="subject" maxlength="100" /></p>
<p><label for="id_message">Message:</label>
    <input type="text" name="message" id="id_message" /></p>
<p><label for="id_sender">Sender:</label>
    <input type="text" name="sender" id="id_sender" /></p>
<p><label for="id_cc_myself">Cc myself:</label>
    <input type="checkbox" name="cc_myself" id="id_cc_myself" /></p>
<input type="submit" value="Submit" />
</form>
```

`as_table`, `as_ul`

Dostosowywanie szablonu formularza

Każde nazwane pole formularza może być użyte w templatce poprzez `{{ form.name_of_field }}` , wygeneruje to kod HTML wyświetlający kontrolkę formularza.

```
1  <form action="/contact/" method="post">
2      <div class="fieldWrapper">
3          {{ form.subject.errors }}
4          <label for="id_subject">E-mail subject:</label>
5          {{ form.subject }}
6      </div>
7      <div class="fieldWrapper">
8          {{ form.message.errors }}
9          <label for="id_message">Your message:</label>
10         {{ form.message }}
11     </div>
12 ...
13     <p><input type="submit" value="Send message" /></p>
14 </form>
```

- `{{ form.name_of_field }}` generuje kontrolkę HTML
- `{{ form.name_of_field.errors }}` wyświetla listę błędów

Formularze dla modeli

ModelForm

Klasa `ModelForm` dziedziczy po klasie `Form` i pozwala na utworzenie formularza odwzorowującego model dajngo

```
>>> from django.forms import ModelForm
>>> from myapp.models import Article

# Create the form class.
>>> class ArticleForm(ModelForm):
...     class Meta:
...         model = Article
...         fields = ['pub_date', 'headline', 'content', 'reporter']

# Creating a form to add an article.
>>> form = ArticleForm()

# Creating a form to change an existing article.
>>> article = Article.objects.get(pk=1)
>>> form = ArticleForm(instance=article)
```

Każde pole modelu będzie reprezentowane przez domyślne pole formularza (pełna lista konwersji jest dostępna w dokumentacji Django)

Metoda `save()`

- Tworzy i zapisuje obiekt do bazy danych z danymi związanymi z formularzem
- Może pobrać kluczowy argument `instance` - jeżeli zostanie podany aktualizuje istniejący obiekt zamiast tworzyć nowy

```
>>> from myapp.models import Article
>>> from myapp.forms import ArticleForm

# Create a form instance from POST data.
>>> f = ArticleForm(request.POST)

# Save a new Article object from the form's data.
>>> new_article = f.save()

# Create a form to edit an existing Article, but use
# POST data to populate the form.
>>> a = Article.objects.get(pk=1)
>>> f = ArticleForm(request.POST, instance=a)
>>> f.save()
```

Metoda `save()`

- Może pobrać argument `commit` - jeżeli ustawiony na `True` (domyślnie), instancja modelu jest zapisywana do bazy danych; jeżeli ustawiony na `False`, tylko tworzy instancję
- Jeżeli `commit=False`, żadne ManyToMany nie mogą być zapisane - instancja jeszcze nie istnieje w bazie danych! Należy użyć później `save_m2m()`.

```
1  # Create a form instance with POST data.
2  >>> f = AuthorForm(request.POST)
3
4  # Create, but don't save the new author instance.
5  >>> new_author = f.save(commit=False)
6
7  # Modify the author in some way.
8  >>> new_author.some_field = 'some_value'
9
10 # Save the new instance.
11 >>> new_author.save()
12
13 # Now, save the many-to-many data for the form.
14 >>> f.save_m2m()
```

Używanie podzbioru pól w formularzu

- Ustaw `editable=False` dla pola modelu aby wyłączyć to pole ze wszystkich formularzy tworzonych dla modelu poprez ModelForm
- Użyj atrybutu `fields` w wewnętrznej klasie `Meta` w `ModelForm` (pozwala również na zmianę kolejności pól)
- Użyj atrybutu `exclude` w wewnętrznej klasie `Meta` w `ModelForm`

```
1  class PartialAuthorForm(ModelForm):
2      class Meta:
3          model = Author
4          fields = ('name', 'title')
5
6  class PartialAuthorForm(ModelForm):
7      class Meta:
8          model = Author
9          exclude = ('birth_date',)
```

Nadpisanie domyślnych typów pól

- Po prostu zadeklaruj pola, których domyślne zachowanie ma być zastąpione.

```
>>> class ArticleForm(ModelForm):
...     pub_date = MyDateFormField()
...
...     class Meta:
...         model = Article
```

- Aby zastąpić kontrolkę, określ parametr `widget` podczas deklaracji pola

```
>>> class ArticleForm(ModelForm):
...     pub_date = DateField(widget=MyDateWidget())
...
...     class Meta:
...         model = Article
```

Form Assets / Form Media

Upiększanie formularzy

- Uczynienie formularza atrakcyjnym i łatwym w użyciu wymaga czegoś więcej niż tylko HTMLa: style CSS i JavaScript
- Użyta kombinacja CSS i JavaScript zależy od konkretnych kontrolek występujących w formularzu
- Django pozwala na przypisanie różnych plików multimedialnych do formularza i kontrolek
- Django Admin definiuje wiele odpowiednio dostosowanych kontrolek, zobacz:
  `django.contrib.admin.widgets`
- Django można zintegrować z dowolną biblioteką JavaScript

Definicje mediów statycznych

Wewnętrzna klasa `Media` może określać wymagania multimedialne.

```
1  class CalendarWidget(forms.TextInput):
2      class Media:
3          css = {
4              'all': ('pretty.css',)
5          }
6          js = ('animations.js', 'actions.js')
```

Statyczne definicje mediów są konwertowane w czasie wykonywania do właściwości (metody) kontrolki `media`.

```
1  >>> w = CalendarWidget()
2  >>> print w.media
3  <link href="http://media.example.com/pretty.css" type="text/css" media="all" rel="
4  <script type="text/javascript" src="http://media.example.com/animations.js">
5  </script>
6  <script type="text/javascript" src="http://media.example.com/actions.js">
7  </script>
```

Opcje `Media`

`css`: słownik opisujący pliki CSS wymagane do różnych form mediów wyjściowych.

- Wartość powinna być krotką/listą plików.
- Klucze powinny być nazwami wyjściowych mediów: all, aural, braille...
- Kluczem może być rozdzielona przecinkami lista typów różnych mediów wyjściowych.

```
1  class Media:
2      css = {
3          'screen': ('pretty.css',),
4          'tv,projector': ('lo_res.css',),
5          'print': ('newspaper.css',)
6      }
```

`js`: krotka opisujące wymagane pliki JavaScript.
UWAGA: Ścieżki do plików multimedialnych będą poprzedzane `settings.MEDIA_URL`.

Media w formularzu

Formularze mogą również posiadać definicje mediów. Są one łączone z definicjami mediów kontrolek:

```
1   class ContactForm(forms.Form):
2       date = DateField(widget=CalendarWidget)
3       name = CharField(max_length=40, widget=OtherWidget)
4
5       class Media:
6           css = {
7               'all': ('layout.css',)
8           }
9
10  >>> f = ContactForm()
11  >>> f.media
12  <link href="http://media.example.com/pretty.css" type="text/css" media="all" rel="
13  <link href="http://media.example.com/layout.css" type="text/css" media="all" rel="
14  <script type="text/javascript" src="http://media.example.com/animations.js"></scri
15  <script type="text/javascript" src="http://media.example.com/actions.js"></script>
16  <script type="text/javascript" src="http://media.example.com/whizbang.js"></script
```

Formsets

Cel

Formset jest warstwą abstrakcji do pracy z wieloma formularzami na tej samej stronie.
Jeśli mamy formularz do tworzenia artykułów...

```
>>> from django import forms
>>> class ArticleForm(forms.Form):
...     title = forms.CharField()
...     pub_date = forms.DateField()
```

...możemy chcieć umożliwić użytkownikom tworzenie wielu artykułów na raz:

```
>>> from django.forms.formsets import formset_factory
>>> ArticleFormSet = formset_factory(ArticleForm)
```

Możemy kontrolować liczbę dodatkowych formularzy i maksymalną liczbę wszystkich formularzy za pomocą argumentów `extra` i `max_num`:

```
>>> ArticleFormSet = formset_factory(ArticleForm, extra=2, max_num=1)
```

Korzystanie z formsets w widokach i templatkach

Tak samo jak w przypadku zwykłej klasy `Form`. Jedyną różnica jest konieczność użycia management form w templatce.

```
1  def manage_articles(request):
2      ArticleFormSet = formset_factory(ArticleForm)
3      if request.method == 'POST':
4          formset = ArticleFormSet(request.POST, request.FILES)
5          if formset.is_valid():
6              # do something with the formset.cleaned_data
7      else:
8          formset = ArticleFormSet()
9      return render(request, 'manage_articles.html', {'formset': formset})
```

`manage_articles.html`:

```
1  <form method="post" action="">
2      {{ formset.management_form }}
3      <table>
4          {% for form in formset.forms %}
5          {{ form }}
6          {% endfor %}
7      </table>
8  </form>
```

Model formsets

Tworzone za pomocą `modelformset_factory`, który jest rozszerzeniem `formset_facotry`:

```
>>> from django.forms.models import modelformset_factory
>>> AuthorFormSet = modelformset_factory(Author)
```

Użyj parametru `queryset` aby ograniczyć liczbę obiektów:

```
>>> formset = AuthorFormSet(queryset=Author.objects.filter(...))
```

Formset'y mają także metodę `save()` która zwraca listę instancji zapisanych w bazie danych:

```
# Create a formset instance with POST data.
>>> formset = AuthorFormSet(request.POST)

# Assuming all is valid, save the data.
>>> instances = formset.save()
```

Przesyłanie plików

Podstawy przesyłania plików

Prosty formularz zawierający pole `FileField`

```
1  from django import forms
2
3  class UploadFileForm(forms.Form):
4      title = forms.CharField(max_length=50)
5      my_file  = forms.FileField()
```

Funkcja widoku obsługująca formularz otrzyma dane pliku w `request.FILES`. Jest to słownik zawierający klucz dla każdego `FileField` (lub `ImageField` lub innej podklasy `FileField`) w formularzu. Tak więc dane z powyższego formularza będą dostępny jako `request.FILES['my_file']`.

Podstawy przesyłania plików (cd)

Uwaga: `request.FILES` będzie zawierał dane tylko w przypadku gdy metodą żądania jest `POST` i `<form>` który przesyła dane ma ustawiony atrybut `enctype="multipart/form-data"`. W przeciwnym wypadku `request.FILES` będzie pusty.

```
1  from django.http import HttpResponseRedirect
2  from django.shortcuts import render
3  from .forms import UploadFileForm
4
5  # Imaginary function to handle an uploaded file.
6  from somewhere import handle_uploaded_file
7
8  def upload_file(request):
9      if request.method == 'POST':
10         form = UploadFileForm(request.POST, request.FILES)
11         if form.is_valid():
12             handle_uploaded_file(request.FILES['my_file'])
13             return HttpResponseRedirect('/success/url/')
14     else:
15         form = UploadFileForm()
16     return render(request, 'upload.html', {'form': form})
```

Obsługa przesłanych plików

```
1  def handle_uploaded_file(f):
2      with open('some/file/name.txt', 'wb+') as destination:
3          for chunk in f.chunks():
4              destination.write(chunk)
```

klasa `UploadedFile`

metody/atrybuty: `read()`, `multiple_chunks()`, `chunks()`, `name`, `size`, `content_type`, `charset`, `temporary_file_path`

Ustawienia kontrolujące przesyłanie plików

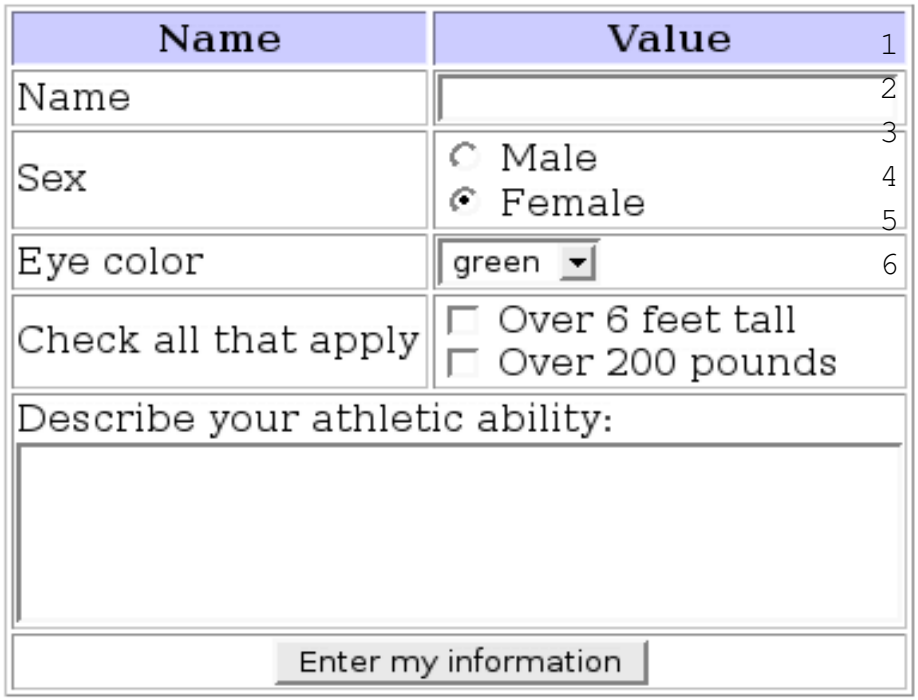
- FILE_UPLOAD_MAX_MEMORY_SIZE - default 2,5 MB
- FILE_UPLOAD_TEMP_DIR - default tmp
- FILE_UPLOAD_PERMISSIONS - default 0x644
- FILE_UPLOAD_HANDLERS

Formularze w RoR

Podstawy

Wprowadzenie do formularzy w RoR

- formularze w aplikacjach web'owych są głównym elementem służącym do pobierania danych od użytkownika
- kontrolki formularzy mogą szybko stać się kłopotliwe w pisaniu i utrzymaniu
- Rails oferuje helpery do generowania kontrolek formularzy
- programista powinie wiedzieć o różnicach pomiędzy kontrolkami




```
<form name="input" action="file_name"
      method="get">
   Username:
   <input type="text" name="user" />
   <input type="submit" value="Submit" />
</form>
```

form_tag

```
1  <% form_tag do %>
2    Zawartość formularza
3  <% end %>
```

form_tag wywołany bez argumentów tworzy element formularza, którego pole "action" odwołuje się do bieżącej strony, poprzez metodę "post"

```
1  <form action="/home/index" method="post">
2    <div style="margin:0;padding:0">
3      <input name="authenticity_token" type="hidden"
4          value="f755bb0ed134b76c432144748a6d4b7a7ddf2b71" />
5    </div>
6    Zawartość formularza
7  </form>
```

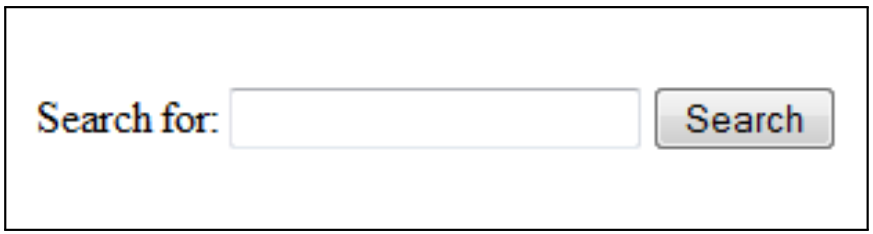
znacznik div, a w nim pole typu "hidden" - forma zabezpieczenia w Railsach o nazwie cross-site request forgery protection – występująca zawsze w przypadku formularza, którego metodą nie jest "get"

Formularze wyszukiwania

```
1  <% form_tag(search_path, :method => "get") do %>
2    <%= label_tag(:q, "Search for:") %>
3    <%= text_field_tag(:q) %>
4    <%= submit_tag("Search") %>
5  <% end %>
```

`form_tag`, `label_tag`, `text_field_tag` i `submit_tag`

```
1  <form action="/search" method="get">
2    <label for="q">Search for:</label>
3    <input id="q" name="q" type="text" />
4    <input name="commit" type="submit" value="Search" />
5  </form>
```



Helpery do generowania elementów formularzy

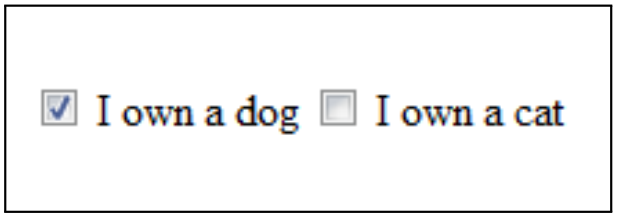
- podstawowe helpery: z nazwą kończącą się na _tag, np. `text_field_tag`, `check_box_tag`, generują pojedynczy element `<input>`
- pierwszym parametrem jest zawsze nazwa pola formularza
- W kontrolerze nazwa ta będzie kluczem w hashu `params` używanym do pobierania wartości wprowadzonej przez użytkownika

```
1  <%= text_field_tag(:query) %> #in view
2  params[:query]                #in coltroller
```

Pola wielokrotnego wyboru - Checkboxes

```
1  <%= check_box_tag(:pet_dog) %>
2    <%= label_tag(:pet_dog, "I own a dog") %>
3  <%= check_box_tag(:pet_cat) %>
4    <%= label_tag(:pet_cat, "I own a cat") %>
```

```
1  <input id="pet_dog" name="pet_dog" type="checkbox" value="1" />
2    <label for="pet_dog">I own a dog</label>
3  <input id="pet_cat" name="pet_cat" type="checkbox" value="1" />
4    <label for="pet_cat">I own a cat</label>
```



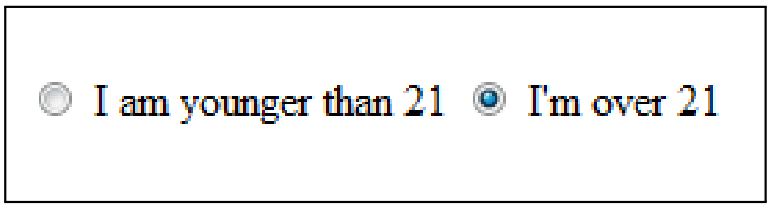
Drugim parametrem `check_box_tag` jest wartość pola formularza (value)

możesz sprawdzić wartość `params[:pet_dog]` i `params[:pet_cat]` by zobaczyć które zwierzęta posiada użytkownik

Pola jednokrotnego wyboru - Radio buttons

```
1  <%= radio_button_tag(:age, "child") %>
2    <%= label_tag(:age_child, "I am younger than 21") %>
3  <%= radio_button_tag(:age, "adult") %>
4    <%= label_tag(:age_adult, "I'm over 21") %>
```

```
1  <input id="age_child" name="age" type="radio" value="child" />
2    <label for="age_child">I am younger than 21</label>
3  <input id="age_adult" name="age" type="radio" value="adult" />
4    <label for="age_adult">I'm over 21</label>
```

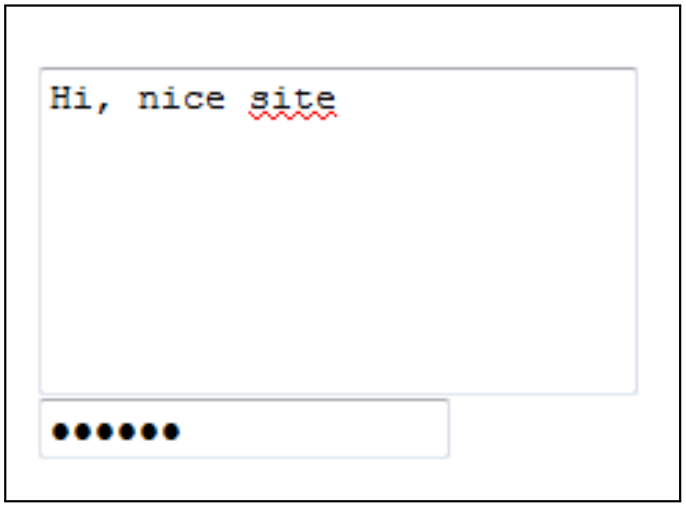


drugim parametrem `radio_button_tag` jest wartość pola formularza `params[:age]` będzie zawierać albo "child", albo "adult"

Pozostałe helpery

```
1  <%= text_area_tag(:message, "Hi, nice site", :size => "24x6") %><br/>
2  <%= password_field_tag(:password) %>
3  <%= hidden_field_tag(:parent_id, "5") %>
```

```
1  <textarea id="message" name="message" cols="24" rows="6">Hi, nice site</textarea><
2  <input id="password" name="password" type="password" />
3  <input id="parent_id" name="parent_id" type="hidden" value="5" />
```



Formularze powiązane z modelami

Helpery modeli obiektowych

- helpery modeli obiektowych - helpery nie posiadające w nazwie końcówki `_tag`, np. `text_field`, `text_area`

```
1 <%= text_field(:person, :name) %>
2 # => <input id="person_name" name="person[name]" type="text" value="Henry"/>
```

- pierwszym atrybutem jest nazwa zmiennej instancji
- drugim atrybutem jest nazwa metody (zwykle atrybut) wywoływanej na tym obiekcie
- Railsy automatycznie wygenerują pole z ustawionym id, nazwą i wartością
- przesłane wartości wprowadzone przez użytkownika będą przechowywane w `params[:person][:name]`
- hash `params[:person]` jest przystosowany do przekazania wartości do `Person.new` lub, jeśli istnieje instancja `@person` klasy `Person`, do `@person.update_attributes`

Przypisanie formularza do obiektu

helper form_for

```
1   #articles_controller.rb
2   def new
3     @article = Article.new
4   end
5
6   #articles/new.html.erb
7   <% form_for :article, @article, :url => { :action => "create" },
8                :html => {:class => "nifty_form"} do |f| %>
9     <%= f.text_field :title %>
10    <%= f.text_area :body, :size => "60x12" %>
11    <%= submit_tag "Create" %>
12  <% end %>
```

```
<form action="/articles/create" method="post" class="nifty_form">
  <input id="article_title" name="article[title]" size="30" type="text" />
  <textarea id="article_body" name="article[body]" cols="60" rows="12"></textarea>
  <input name="commit" type="submit" value="Create" />
</form>
```

Metoda `form_for` tworzy obiekt `form builder` (zmienna `f`)

Helper `fields_for`

- helper `fields_for` - nie tworzy tagów <form>
- przydatne podczas edycji dodatkowych modeli obiektowych tego samego formularza

```
1  <% form_for @person do |person_form| %>
2    <%= person_form.text_field :name %>
3    <% fields_for @person.contact_detail do |contact_details_form| %>
4      <%= contact_details_form.text_field :phone_number %>
5    <% end %>
6  <% end %>
```

```
<form action="/people/1" class="edit_person" id="edit_person_1" method="post">
  <input id="person_name" name="person[name]" size="30" type="text" />
  <input id="contact_detail_phone_number"
         name="contact_detail[phone_number]" size="30" type="text" />
</form>
```

Formularze z metodami PUT lub DELETE

- większość przeglądarek nie obsługuje metod do wysyłania formularzy innych niż "GET" i "POST"
- w Railsach problem ten rozwiązany jest w taki sposób, że inne metody naśladowane są przez POST, a ukryte pole o nazwie `_method`, odzwierciedla wybraną metodę

```
1   form_tag(search_path, :method => "put")
```

```
<form action="/search" method="post">
  <div style="margin:0;padding:0">
    <input name="_method" type="hidden" value="put" />
    <input name="authenticity_token" type="hidden"
           value="f755bb0ed134b76c432144748a6d4b7a7ddf2b71" />
  </div>
  ...
```

Pola typu select

Tagi `select_tag` i `option`

```
1  <select name="city_id" id="city_id">
2    <option value="1">Lisbon</option>
3    <option value="2">Madrid</option>
4    ...
5    <option value="12">Berlin</option>
6  </select>
```

- `select_tag`

```
1  <%= select_tag(:city_id, '<option value="1">Lisbon</option>...') %>
```

- `options_for_select`

```
1  <%= options_for_select([['Lisbon', 1], ['Madrid', 2], ...],2) %>
2  # => <option value="1">Lisbon</option>
3  # => <option value="2" selected="selected">Madrid</option>
4  # => ...
```

- połączenie `select_tag` i `options_for_select`

```
1  <%= select_tag(:city_id, options_for_select(...)) %>
```

Pola typu select do operowania na obiektach

- helper `select`

```
1  # controller:
2  @person = Person.new(:city_id => 2)
3
4  # view:
5  <%= select(:person, :city_id, [['Lisabon', 1], ['Madrid', 2], ...]) %>
6  # select on a form builder
7  <%= f.select(:city_id, ...) %>
```

Tagi opcji jako kolekcje dowolnych obiektów

- stworzenie zagnieżdzonej tablicy poprzez wyliczenie iteracyjne obiektów

```
1  <% cities_array = City.find(:all).map { |city| [city.name, city.id] } %>
2  <%= options_for_select(cities_array) %>
```

- helper `options_from_collection_for_select`

```
1  <%= options_from_collection_for_select(City.all, :id, :name) %>
```

- helper `collection_select`

```
1  <%= collection_select(:person, :city_id, City.all, :id, :name) %>
```

Helpery do obsługi pól data i czas

Helpery do obsługi pól data i czas

- daty i godziny nie są wyrażane poprzez pojedynczy element formularza
- helpery niezależne: `select_date`, `select_time` i `select_datetime`

```
1  <%= select_date Date::today, :prefix => :start_date %>
```

```
1 <select id="start_date_year" name="start_date[year]"> ... </select>
2 <select id="start_date_month" name="start_date[month]"> ... </select>
3 <select id="start_date_day" name="start_date[day]"> ... </select>
```

- helpery modelu obiektowego: `date_select`, `time_select` i `datetime_select`

```
1  <%= date_select :person, :b_date %>
```

```
1 <select id="person_b_date_1i" name="person[b_date(1i)]"> ... </select>
2 <select id="person_b_date_2i" name="person[b_date(2i)]"> ... </select>
3 <select id="person_b_date_3i" name="person[b_date(3i)]"> ... </select>
```

Przesyłanie plików

Uploading files

- kodowanie formularza MUSI być ustawione na "multipart/form-data"

```
1  <% form_tag({:action => :upload}, :multipart => true) do %>
2    <%= file_field_tag 'picture' %>
3  <% end %>
4
5  <% form_for @person, :html => {:multipart => true} do |f| %>
6    <%= f.file_field :picture %>
7  <% end %>
```

- Obiekt w hashu `params` jest instancją podklasy `IO`. W zależności do wielkości załadowanego pliku może to być `StringIO` lub instancja `File` z tymczasowym plikiem.

```
1  def upload
2    uploaded_io = params[:person][:picture]
3    File.open(Rails.root.join('public', 'uploads',
4              uploaded_io.original_filename), 'w') do |file|
5      file.write(uploaded_io.read)
6    end
7  end
```

Wyświetlanie błędów walidacji

Helpery do walidacji

- helpery do walidacji używane są w definicji klasy modelu
- helpery do walidacji -> reguły walidacji
- kolekcja `errors` obiektu
- atrybuty helperów, opcje `:on` (:save (the default), :create or :update) i `:message`

```
1  class Person < ActiveRecord::Base
2    validates_acceptance_of :terms_of_service
3    validates_confirmation_of :email
4    validates_presence_of :email_confirmation
5    validates_format_of :description, :with => /^[a-zA-Z]+$/,
6        :message => "Only letters allowed"
7    validates_length_of :name, :minimum => 2
8    validates_uniqueness_of :email
9  end
```

Użycie kolekcji `errors` w pliku widoku I

metoda `error_messages`

```
1  class Product < ActiveRecord::Base
2    validates_presence_of :description, :value
3    validates_numericality_of :value, :allow_nil => true
4  end
```

```
1  <% form_for(@product) do |f| %>
2    <%= f.error_messages %>
3    <p>
4      <%= f.label :description %><br />
5      <%= f.text_field :description %>
6    </p>
7    <p>
8      <%= f.label :value %><br />
9      <%= f.text_field :value %>
10   </p>
11   <p>
12     <%= f.submit "Create" %>
13   </p>
14 <% end %>
```

Użycie kolekcji `errors` w pliku widoku II