DJANGO FORM

江琳

HTML FORM

- **In HTML, a form is a collection of elements inside <form>…</form> that allow a visitor to do things like enter text, select options, manipulate objects or controls, and so on, and then send that information back to the server.**

HTML FORM

- **Do you remember the form we write for backstage login page in index.html?**

```
<form method="post">
    {% csrf_token %}
    <p style="height: 15px">
        {% for msg in messages %}
        <span class="{{ msg.tags }}">{{ msg }}</span>
        {% endfor %}
    </p>
    <p>
        <label>Username:</label>
        <input type="text" placeholder="username"
name="username">
    </p>
    <p>
       <label>Password:</label>
       <input type="password" placeholder="password"
name="password">
    </p>
    <input type="submit" value="Submit">
</form>
```

FORM

DJANGO FORM

- **Django's form is simplify and automate, and it more securely than most programmers would be able to do in code they wrote themselves.**

DJANGO FORM

- **Let's rewrite the backstage login page using Django Form.**
- **Firstly, we need create a file named forms.py in the application of books.**

THE FROMS

- We already know what we want our HTML form to look like. Our starting point for it in Django is this:

```
from django import forms

class LoginForm(forms.Form):
    username = forms.CharField(widget=forms.TextInput(
        attrs={'placeholder': 'username'}))
    password = forms.CharField(widget=forms.PasswordInput(
        attrs={'placeholder': 'password'}
    ))
```

THE FROMS

- **This defines a Form class with two fields username and password**

THE VIEWS

```
from .forms import LoginForm
class Index(View):

    def get(self, request):
        form = LoginForm
        return render(request, ‘backstage/index.html’, {‘form’:form})

    def post(self, request):
        form = LoginForm(request.POST)
        if form.is_valid():
            user = authenticate(
                username=form.cleaned_data[‘username’],
                password=form.cleaned_data[‘password’])
            if user and user.is_superuser:
                login(request, user)
                return redirect(reverse(‘bs_list’))
        else:
            messages.error(request, ‘username or password is incorrect！’)
            return redirect(reverse(‘index’))
        return render(request, 'backstage/index.html', {'form': form})
```

THE TEMPLATE

- **Now we don't need to do much in index.html**

```
<form method="post">
    {% csrf_token %}
    <p>
    {% for msg in messages %}
        <span class="{{ msg.tags }}">{{ msg }}</span>
    {% endfor %}
    </p>
    {{ form.as_p }}
    <input type="submit" value="Submit">
</form>
```

DJANGO FORM

- **If we arrive at this view with a GET request, it will create an empty form instance and place it in the template context to be rendered.**

DJANGO FORM

- **If the form is submitted using a POST request, the view will once again create a form instance and populate it with data from the request:**
  - **form = LoginForm(request.POST)**
- **This is called "binding data to the form"**

DJANGO FORM

- A Form instance has an is_valid() method, which runs validation routines for all its fields. When this method is called, if all fields contain valid data, it will:
  - return True
  - place the form's data in its cleaned_data attribute.
- it's not True, we go back to the template with the form.

DJANGO FORM

- **Assume we limit the length of username is 16 in maximum.**

```
class LoginForm(forms.Form):
    username = forms.CharField(widget=forms.TextInput(
        attrs={'placeholder': 'username'}))
    password = forms.CharField(widget=forms.PasswordInput(
        attrs={'placeholder': 'password'}
    ))

    def clean_username(self):
        username = self.cleaned_data['username']
        if len(username) > 16:
            raise ValidationError('username is to long')
        return username
```

DJANGO FORM

- **Form.is_vaild() will return False if length of username is longer than 16.**

BUILD-IN FIELD CLASSES

- **When you create a Form class, the most important part is defining the fields of the form.**

```
from django import forms

class LoginForm(forms.Form):
    username =
forms.CharField(widget=forms.TextInput(
        attrs={'placeholder': 'username'}))
    password =
forms.CharField(widget=forms.PasswordInput(
        attrs={'placeholder': 'password'}
    ))
```

BUILD-IN FIELD CLASSES

- https://docs.djangoproject.com/en/1.11/ref/forms/fields/
- **BooleanField**
- **CharField**
- **ChoiceField**
- **DateTimeField**
- **FileField**

BUILD-IN FIELD WIDGETS

- A widget is Django's representation of an HTML input element. The widget handles the rendering of the HTML, and the extraction of data from a GET/POST dictionary that corresponds to the widget.

BUILD-IN FIELD WIDGETS

- https://docs.djangoproject.com/en/1.11/ref/forms/widgets/

```
from django import forms

class LoginForm(forms.Form):
    username =
forms.CharField(widget=forms.TextInput(
        attrs={'placeholder': 'username'}))
    password =
forms.CharField(widget=forms.PasswordInput(
        attrs={'placeholder': 'password'}
    ))
```

MODELFORM

MODELFORM

- If you're building a database-driven app, chances are you'll have forms that map closely to Django models.
- For instance, we have Book model in our Library Management System, and we want to create a form that lets administrator create new book.

MODELFORM

- For this reason, Django provides a helper class that lets you create a Form class from a Django model.
- Do you remember we have a CreateAuthor view used to create author of book? Lets modify it.

THE VIEW

```
class CreateAuthor(CreateView):
    form class = AuthorForm
    template_name =
'backstage/create_author.html'

    def get_success_url(self):
        messages.success(self.request, 'Create
Success!')
        return resolve_url('create_author')
```

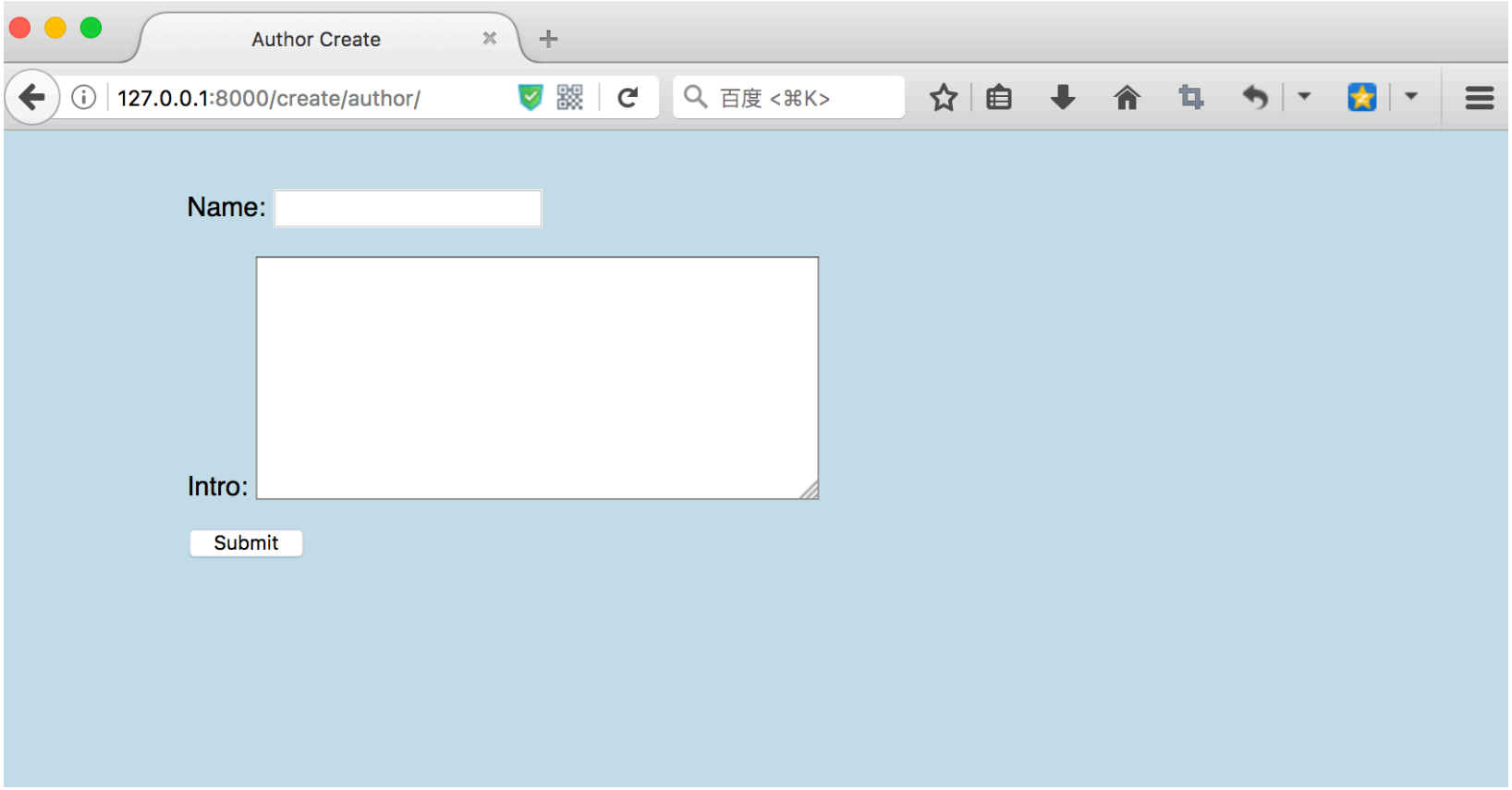
THE FORM

- In forms.py

```
from django.forms import ModelForm
from .models import Author

class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = '__all__'
```

Set the fields attribute to the special value '__all__' to indicate that all fields in the model should be used.

MODELFORM


Author Create
127.0.0.1:8000/create/author/
百度 <⌘K>
Name:
Intro:
Submit

VALIDATION

- **Model validation is triggered after the form's clean() method is called.**
- **You can override the clean() method on a model form to provide additional validation.**
- **Let write a clean() method to avoid user to create repeated author.**

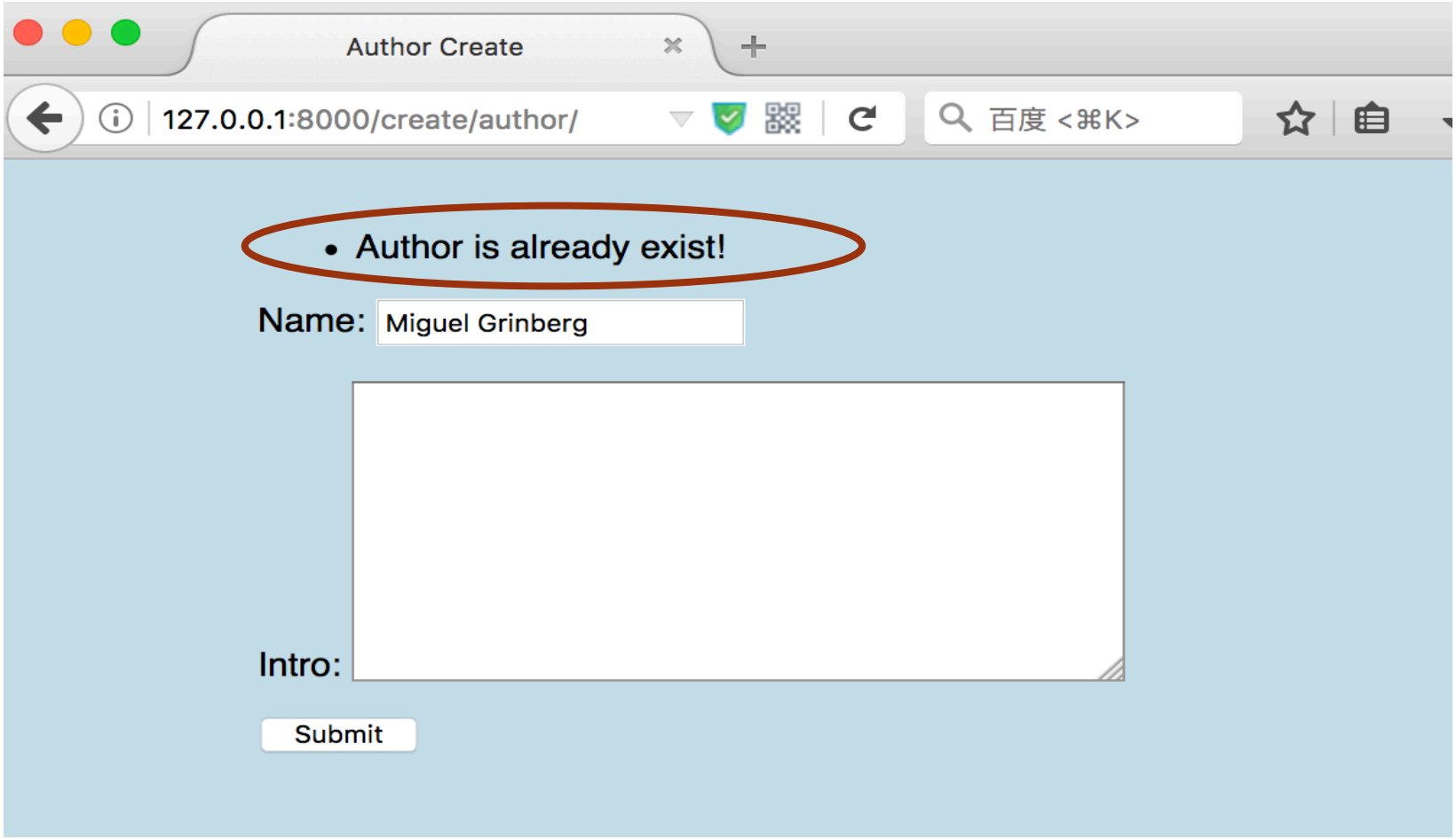
VALIDATION

```
from django.forms import ModelForm, ValidationError
from .models import Author

class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = '__all__'

    def clean_name(self):
        name = self.cleaned_data['name']
        if Author.objects.filter(name=name).exists():
            raise ValidationError('Author is already
exist!')
        return name
```

VALIDATION

- **Now if we try to create a user which already exist, a error message will show up.**

VALIDATION


Author Create
127.0.0.1:8000/create/author/
百度 <⌘K>
• Author is already exist!
Name:
Miguel Grinberg
Intro:
Submit

REFERENCE

- https://docs.djangoproject.com/en/1.11/topics/forms/
- https://docs.djangoproject.com/en/1.11/ref/forms/fields/
- https://docs.djangoproject.com/en/1.11/ref/forms/widgets/
- https://docs.djangoproject.com/en/1.11/topics/forms/modelforms/

Questions?