Django and AJAX Form Submissions – Say 'Goodbye' to the Page Refresh

by Real Python 70 Comments django front-end python

Table of Contents

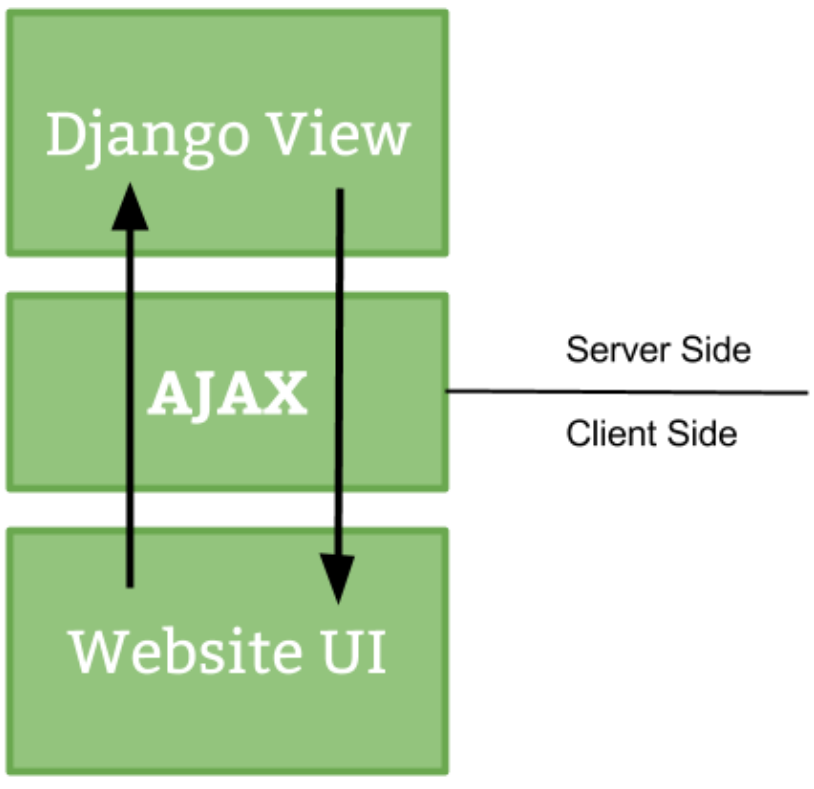
Let's get down to business:

1. Download the compressed pre-ajax Django Project from the repo
2. Activate a virtualenv
3. Install the requirements
4. Sync the database
5. Fire up the server

Once logged in, test out the form. What we have here is a *simple* communication app with just create rights. It looks nice, but there's one annoying issue: The page refresh.

How do we get rid of it? Or, how do we update just a *portion* of a webpage without having to refresh the *entire* page?

Enter AJAX. AJAX is a client-side technology used for making asynchronous requests to the server-side - i.e., requesting or submitting data - where the subsequent responses do not cause an entire page refresh.



This tutorial assumes you have working knowledge of Django as well as **some** experience with JavaScript/jQuery. You should also be familiar with the basic HTTP methods, particularly GET and POST. Need to get up to speed? Get Real Python.

Free Bonus: Click here to get access to a free Django Learning Resources Guide (PDF) that shows you tips and tricks as well as common pitfalls to avoid when building Python + Django web applications.

This is a collaboration piece between Real Python and the mighty Nathan Nichols, using a collaborative method we have dubbed "agile blogging". Say "hi" to Nathan on Twitter: @natsamnic.

Use Protection

Regardless of whether you're using AJA

> Read more about CSRF attacks on th

To prevent such attacks, you must add t
field containing a token that gets sent w

If you look at the *talk/index.html* templa
comes to AJAX requests, we need to add
object since the scripts are static.

To get around this, we need to create a custom header that includes the token to watch our back. Simply grab the code here and add it to the end of the *main.js* file. Yes, it's a lot of code. We could go through it line-by-line, but that's not the point of this post. Just trust us that it works.

Moving on…

Handling Events

Before we touch the AJAX code, we need to add an event handler to our JavaScript file using jQuery.

> Keep in mind that jQuery is JavaScript. It's simply a JavaScript library used to reduce the amount of code you need to write. This is a common area of confusion so just be mindful of this as you go through the remainder of this tutorial.

Which event(s) do we need to "handle"? Since we're just working with creating a post at this point, we just need to add one handler to *main.js*:

JavaScript

```
// Submit post on submit
$('#post-form').on('submit', function(event){
    event.preventDefault();
    console.log("form submitted!")  // sanity check
    create_post();
});
```

Here, when a user submits the form this function fires, which-

1. Prevents the default browser behavior for a form submission,
2. Logs "form submitted!" to the console, and
3. Calls a function called `create_post()` where the AJAX code will live.

Make sure to add an id of `post-form` to the form on the *index.html* file:

HTML

```
<form action="/create_post/" method="POST" id="post-form">
```

And add a link to the JavaScript file to the bottom of the template:

HTML

```
<script src="static/scripts/main.js"></script>
```

Test this out. Fire up the server, then open your JavaScript console. You should see the following when you submit the form:

```
form submitted!
Uncaught ReferenceError: create_post is not defined
```

This is exactly what we should see: The f[illegible] and the `create_post` function is called

Adding AJAX

Let's develop one last iteration before w[illegible]

Update *main.js*:

Add the `create_post` function:

JavaScript

```
// AJAX for posting
function create_post() {
    console.log("create post is working!") // sanity check
    console.log($('#post-text').val())
};
```

Again, we ran a sanity check to ensure the function is called correctly, then we grab the input value of the form. For this to work correctly we need to add an id to the form field:

Update *forms.py*:

Python

```
class PostForm(forms.ModelForm):
    class Meta:
        model = Post
        # exclude = ['author', 'updated', 'created', ]
        fields = ['text']
        widgets = {
            'text': forms.TextInput(attrs={
                'id': 'post-text',
                'required': True,
                'placeholder': 'Say something...'
            }),
        }
```

Notice how we also added a placeholder to the field and made it required along with the id. We could add some error handlers to the form template or simply let HTML5 handle it. Let's use the latter.

Test again. Submit the form with the word "test". You should see the following in your console:

```
form submitted!
create post is working!
test
```

Sweet. So, we've confirmed that we're calling the `create_post()` function correctly as well as grabbing the value of the form input. Now let's wire in some AJAX to submit the POST request.

Update *main.js*:

JavaScript

```
// AJAX for posting
function create_post() {
    console.log("create post is working!") // sanity check
    $.ajax({
        url : "create_post/", // the endpoint
        type : "POST", // http method
        data : { the_post : $('#post-text').val() }, // data sent with the post request

        // handle a successful response
        success : function(json) {
            $('#post-text').val(''); // remove the value from the input
            console.log(json); //
            console.log("success"
        },

        // handle a non-successfu
        error : function(xhr,errm
            $('#results').html("<
encountered an error: "+errmsg+
                " <a href='#' cla
            console.log(xhr.statu
the error to the console
        }
    });
};
```


What's happening? Well, we submit form data to the `create_post/` endpoint, then wait for one of two responses - either a success or a failure.... Follow the code comments for a more detailed explanation.

Update the views

Now let's update our views to handle the POST request correctly:

Python

```
def create_post(request):
    if request.method == 'POST':
        post_text = request.POST.get('the_post')
        response_data = {}

        post = Post(text=post_text, author=request.user)
        post.save()

        response_data['result'] = 'Create post successful!'
        response_data['postpk'] = post.pk
        response_data['text'] = post.text
        response_data['created'] = post.created.strftime('%B %d, %Y %I:%M %p')
        response_data['author'] = post.author.username

        return HttpResponse(
            json.dumps(response_data),
            content_type="application/json"
        )
    else:
        return HttpResponse(
            json.dumps({"nothing to see": "this isn't happening"}),
            content_type="application/json"
        )
```

Here we grab the post text along with the author and update the database. Then we create a response dict, serialize it into JSON, and then send it as the response - which gets logged to the console in the success handler: `console.log(json)`, as you saw in the `create_post()` function in the JavaScript file above.

Test this again.

You should see the object in the console:

```
form submitted!
create post is working!
Object {text: "hey!", author: "michael", postpk: 15, result: "Create post successful!", created:
"August 22, 2014 10:55 PM"}
success
```

How about we add the JSON to the DOM!

Updating the DOM

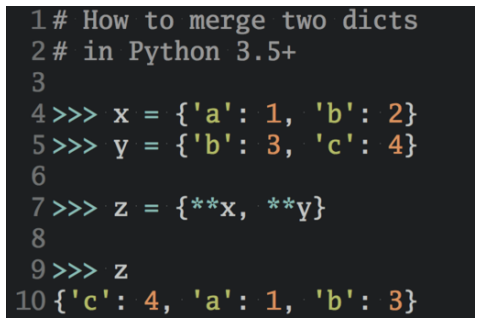
Update the template

Simply add an id of “talk” to the `<ul>`:

HTML

```
<ul id="talk">
```

Then update the form so that errors will



HTML

```
<form method="POST" id="post-form">
    {% csrf_token %}
    <div class="fieldWrapper" id="the_post">
        {{ form.text }}
    </div>
    <div id="results"></div> <!-- errors go here -->
    <input type="submit" value="Post" class="tiny button">
</form>
```

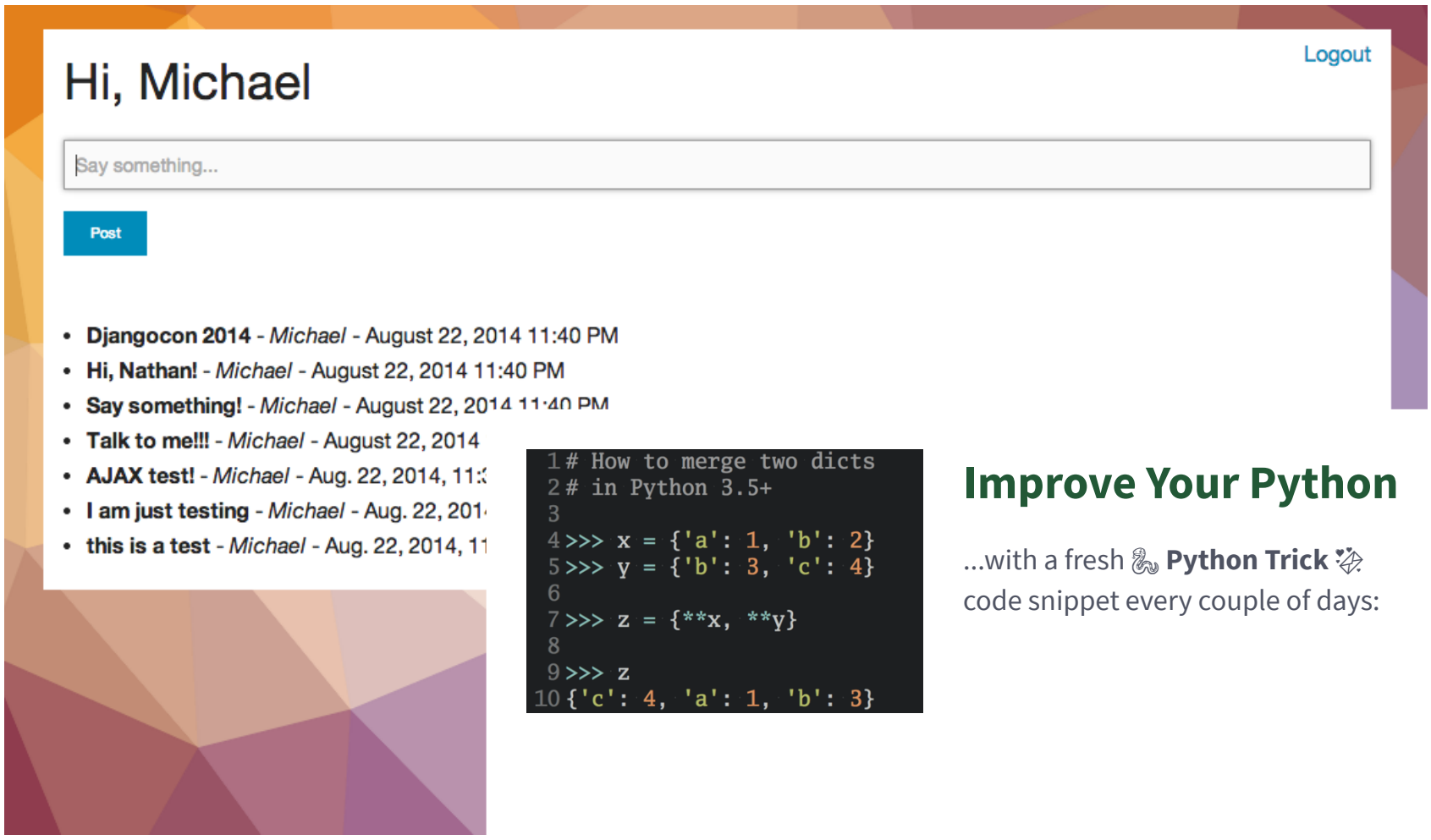
Update *main.js*

Now we can add the JSON to the DOM where that new “talk” id is:

JavaScript

```
success : function(json) {
    $('#post-text').val(''); // remove the value from the input
    console.log(json); // log the returned json to the console
    $("#talk").prepend("<li><strong>"+json.text+"</strong> - <em> "+json.author+"</em> - <span>
"+json.created+"</span></li>");
    console.log("success"); // another sanity check
},
```

Ready to see this in action? Test it out!



If you'd like to see what an error looks like, then comment out all the CSRF Javascript in *main.js* and then try to submit the form.

Rinse, Repeat

Your turn. We need to handle some more events. With your new found knowledge of jQuery and AJAX, you get to put these into place. I added code to the final app - which you can download here - that includes a delete link. You just need to add an event to handle the click, which then calls a function that uses AJAX to send a POST request to the back-end to delete the post from the database. Follow the same workflow as I did in this tutorial. We'll post the answer to this next time.

If you get stuck, and can't debug the errors, follow this workflow -

1. Use the "Google-it-first" algorithm
2. Struggle. Spin your wheels. Set the code aside. Run around the block. Then come back to it.
3. Still stuck? Comment below, stating first the problem and then detailing the steps you've taken to solve the problem thus far

Be sure to *try* troubleshooting on your own before asking for help. Spinning your wheels, hacking away at a solution will benefit you in the long run. It's the process that matters, not so much the solution. It's part of what separates poor developers from great developers. Good luck.

Check out the solution here.

Conclusion

How does your app look? Ready for more?

1. AJAX is *so* yesterday. We can do a lot more with much less code using AngularJS.
2. In most cases, it's a standard to couple the client-side JavaScript, whether it's AJAX or Angular or some other framework, with a server-side RESTful API.
3. Where's the tests?

What would you like to see next? Comment below. Cheers!

Happy coding!