By **Vitor Freitas**

I'm a passionate software developer and researcher from Brazil, currently living in Finland. I write about Python, Django and Web Development on a weekly basis. Read more.

TUTORIAL

How to Implement CRUD Using Ajax and Json

Nov 15, 2016 35 minutes read 71 comments 22,696 views

(Picture: https://www.pexels.com/photo/macbook-laptop-smartphone-apple-7358/)

Using Ajax to create asynchronous request to manipulate Django models is a very common use case. It can be used to provide an in line edit in a table, or create a new model instance without going back and forth in the website. It also bring some challenges, such as keeping the state of the objects consistent.

In case you are not familiar with the term CRUD, it stand for **C**reate **R**ead **U**pdate **D**elete.

Those are the basic operations we perform in the application entities. For the most part the Django Admin is all about CRUD.

This tutorial is compatible with Python 2.7 and 3.5, using Django 1.8, 1.9 or 1.10.

Table of Contents

Basic Configuration

For this tutorial we will be using jQuery to implement the Ajax requests. Feel free to use any other JavaScript framework (or to implement it using bare JavaScript). The concepts should remain the same.

Grab a copy of jQuery, either download it or refer to one of the many CDN options.

jquery.com/download/

I usually like to have a local copy, because sometimes I have to work off-line. Place the jQuery in the bottom of your base template:

base.html

```
{% load static %}<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Bookstore - Simple is Better Than Complex</title>
    <link href="{% static 'css/bootstrap.min.css' %}" rel="stylesheet">
    <!--[if lt IE 9]>
      <script src="https://oss.maxcdn.com/html5shiv/3.7.3/html5shiv.min.js"></script>
      <script src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js"></script>
    <![endif]-->
  </head>
  <body>
    {% include 'includes/header.html' %}
    <div class="container">
      {% block content %}
      {% endblock %}
    </div>
    <script src="{% static 'js/jquery-3.1.1.min.js' %}"></script>  <!-- JQUERY HERE -->
    <script src="{% static 'js/bootstrap.min.js' %}"></script>
    {% block javascript %}
    {% endblock %}
  </body>
</html>
```

I will be also using Bootstrap. It is not required but it provide a good base css and also some useful HTML components, such as a Modal and pretty tables.

Working Example

I will be working in a app called **books.** For the CRUD operations consider the following model:

models.py

```
class Book(models.Model):
    HARDCOVER = 1
    PAPERBACK = 2
    EBOOK = 3
    BOOK_TYPES = (
        (HARDCOVER, 'Hardcover'),
        (PAPERBACK, 'Paperback'),
        (EBOOK, 'E-book'),
    )
    title = models.CharField(max_length=50)
    publication_date = models.DateField(null=True)
    author = models.CharField(max_length=30, blank=True)
    price = models.DecimalField(max_digits=5, decimal_places=2)
    pages = models.IntegerField(blank=True, null=True)
    book_type = models.PositiveSmallIntegerField(choices=BOOK_TYPES)
```

Listing Books

Let’s get started by listing all the book objects.

We need a route in the urlconf:

urls.py:

```
from django.conf.urls import url, include
from mysite.books import views

urlpatterns = [
    url(r'^books/$', views.book_list, name='book_list'),
]
```

A simple view to list all the books:

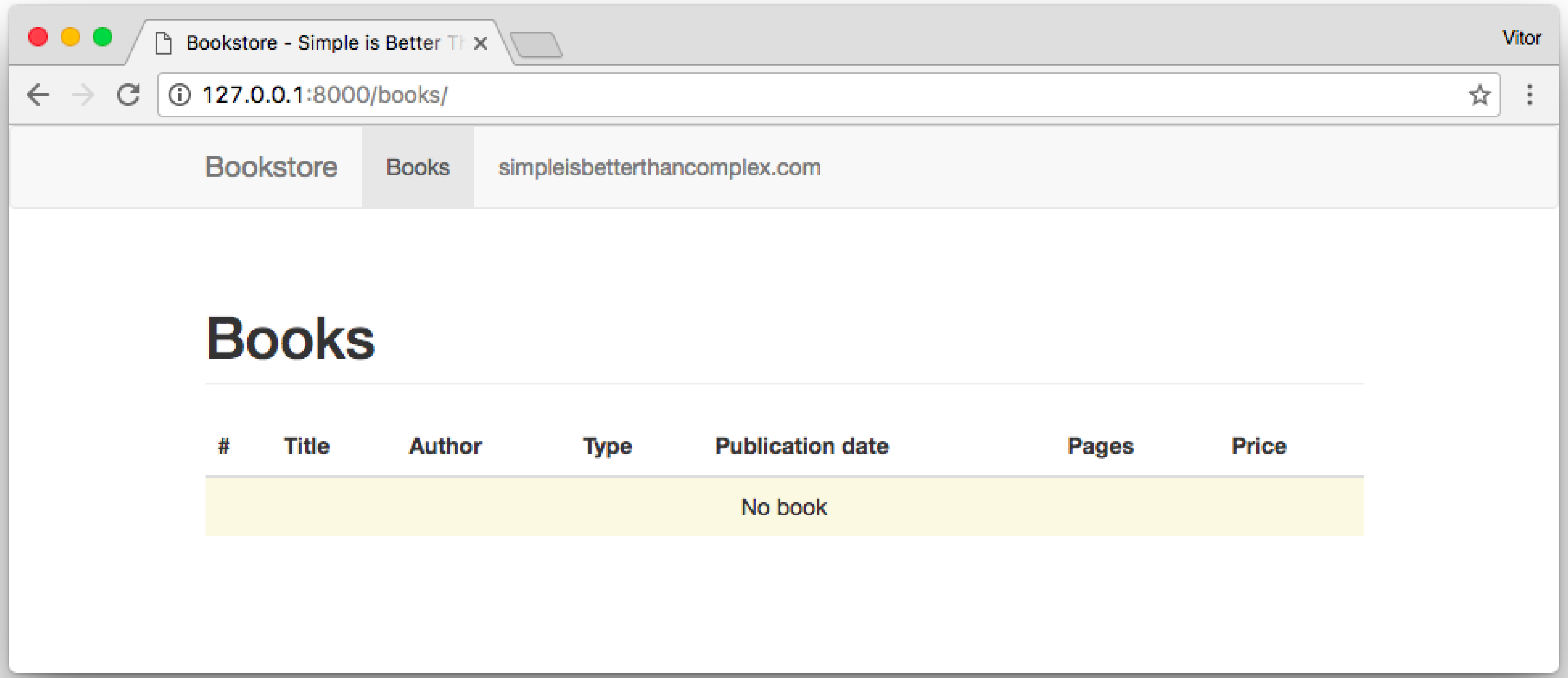
views.py

```
from django.shortcuts import render
from .models import Book

def book_list(request):
    books = Book.objects.all()
    return render(request, 'books/book_list.html', {'books': books})
```

book_list.html

```
{% extends 'base.html' %}

{% block content %}
  <h1 class="page-header">Books</h1>
  <table class="table" id="book-table">
    <thead>
      <tr>
        <th>#</th>
        <th>Title</th>
        <th>Author</th>
        <th>Type</th>
        <th>Publication date</th>
        <th>Pages</th>
        <th>Price</th>
      </tr>
    </thead>
    <tbody>
      {% for book in book_list %}
        <tr>
          <td>{{ book.id }}</td>
          <td>{{ book.title }}</td>
          <td>{{ book.author }}</td>
          <td>{{ book.get_book_type_display }}</td>
          <td>{{ book.publication_date }}</td>
          <td>{{ book.pages }}</td>
          <td>{{ book.price }}</td>
        </tr>
      {% empty %}
        <tr>
          <td colspan="7" class="text-center bg-warning">No book</td>
        </tr>
      {% endfor %}
    </tbody>
  </table>
{% endblock %}
```

So far nothing special. Our template should look like this:


Bookstore - Simple is Better Th
Vitor
127.0.0.1:8000/books/
Bookstore
Books
simpleisbetterthancomplex.com

Books

#	Title	Author	Type	Publication date	Pages	Price
			No book			

Create Book

First thing, let’s create a model form. Let Django do its work.

forms.py

```
from django import forms
from .models import Book

class BookForm(forms.ModelForm):
    class Meta:
        model = Book
        fields = ('title', 'publication_date', 'author', 'price', 'pages', 'book_type', )
```

We need now to prepare the template to handle the creation operation. We will be working with partial templates to render only the parts that we actually need.

The strategy I like to use is to place a generic bootstrap modal, and use it for all the operations.

book_list.html

```
{% extends 'base.html' %}

{% block content %}
  <h1 class="page-header">Books</h1>

  <!-- BUTTON TO TRIGGER THE ACTION -->
  <p>
    <button type="button" class="btn btn-primary js-create-book">
      <span class="glyphicon glyphicon-plus"></span>
      New book
    </button>
  </p>

  <table class="table" id="book-table">
    <!-- TABLE CONTENT SUPPRESSED FOR BREVITY'S SAKE -->
  </table>

  <!-- THE MODAL WE WILL BE USING -->
  <div class="modal fade" id="modal-book">
    <div class="modal-dialog">
      <div class="modal-content">
      </div>
    </div>
{% endblock %}
```

Note that I already added a **button** that will be used to start the creation process. I added a class **js-create-book** to hook the click event. I usually add a class starting with **js-** for all elements that interacts with JavaScript code. It's easier to debug the code later on. It's not a enforcement but just a convention. Helps the code quality.

Add a new route:

urls.py:

```
from django.conf.urls import url, include
from mysite.books import views

urlpatterns = [
    url(r'^books/$', views.book_list, name='book_list'),
    url(r'^books/create/$', views.book_create, name='book_create'),
]
```

Let’s implement the **book_create** view:

views.py

```
from django.http import JsonResponse
from django.template.loader import render_to_string
from .forms import BookForm

def book_create(request):
    form = BookForm()
    context = {'form': form}
    html_form = render_to_string('books/includes/partial_book_create.html',
        context,
        request=request,
    )
    return JsonResponse({'html_form': html_form})
```

Note that we are not rendering a template but returning a Json response.

Now we create the partial template to render the form:

partial_book_create.html

```
{% load widget_tweaks %}
```

```
<form method="post">
  {% csrf_token %}
  <div class="modal-header">
    <button type="button" class="close" data-dismiss="modal" aria-label="Close">
      <span aria-hidden="true">&times;</span>
    </button>
    <h4 class="modal-title">Create a new book</h4>
  </div>
  <div class="modal-body">
    {% for field in form %}
      <div class="form-group{% if field.errors %} has-error{% endif %}">
        <label for="{{ field.id_for_label }}">{{ field.label }}</label>
        {% render_field field class="form-control" %}
        {% for error in field.errors %}
          <p class="help-block">{{ error }}</p>
        {% endfor %}
      </div>
    {% endfor %}
  </div>
  <div class="modal-footer">
    <button type="button" class="btn btn-default" data-dismiss="modal">Close</button>
    <button type="submit" class="btn btn-primary">Create book</button>
  </div>
</form>
```

I'm using the **django-widget-tweaks** library to render the form fields properly using the bootstrap class. You can read more about it in a post I published last year: Package of the Week: Django Widget Tweaks.

Now the glue that will put everything together: JavaScript.

Create an external JavaScript file. I created mine in the path: **mysite/books/static/books/js/books.js**

books.js

```
$(function () {

  $(".js-create-book").click(function () {
    $.ajax({
      url: '/books/create/',
      type: 'get',
      dataType: 'json',
      beforeSend: function () {
        $("#modal-book").modal("show");
      },
      success: function (data) {
        $("#modal-book .modal-content").html(data.html_form);
      }
    });
  });

});
```

Don’t forget to include this JavaScript file in the **book_list.html** template:

book_list.html

```
{% extends 'base.html' %}

{% load static %}

{% block javascript %}
  <script src="{% static 'books/js/books.js' %}"></script>
{% endblock %}

{% block content %}
  <!-- BLOCK CONTENT SUPPRESSED FOR BREVITY'S SAKE -->
{% endblock %}
```

Let’s explore the JavaScript snippet in great detail:

This is a jQuery shortcut to tell the browser to wait for all the HTML be rendered before executing the code:

```
$(function () {
  ...
});
```

Here we are hooking into the click event of the element with class **js-create-book**, which is our Add book button.

```
$(".js-create-book").click(function () {
  ...
});
```

When the user clicks in the **js-create-book** button, this anonymous function with the **$.ajax** call will be executed:

```
$.ajax({
  url: '/books/create/',
  type: 'get',
  dataType: 'json',
  beforeSend: function () {
    $("#modal-book").modal("show");
  },
  success: function (data) {
    $("#modal-book .modal-content").html(data.html_form);
  }
});
```

Now, what is this ajax request saying to the browser:

Hey, the resource I want is in this path:

```
url: '/books/create/',
```

Make sure you request my data using the HTTP GET method:

```
type: 'get',
```

Oh, by the way, I want to receive the data in JSON format:

```
dataType: 'json',
```

But just before you communicate with the server, please execute this code:

```
beforeSend: function () {
  $("#modal-book").modal("show");
},
```

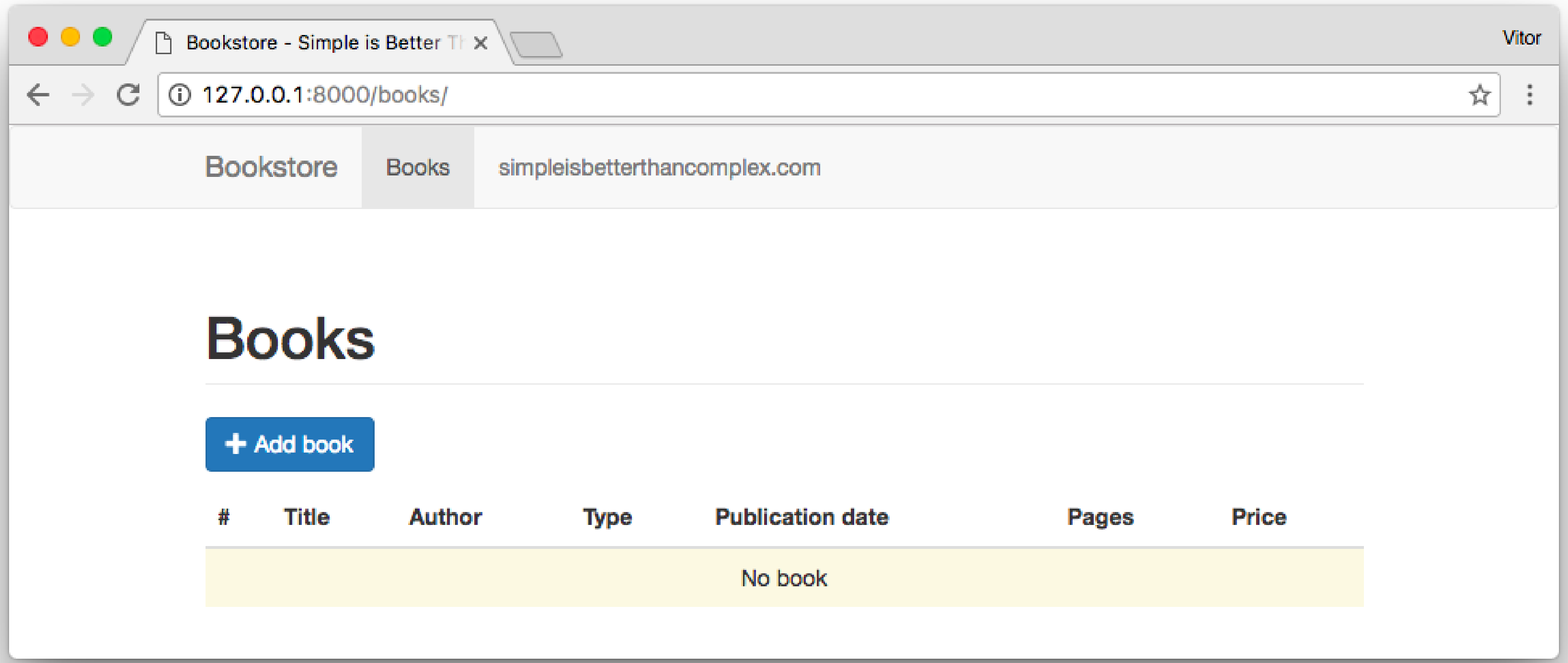
(This will open the Bootstrap Modal before the Ajax request starts.)

And right after you receive the data (in the data variable), execute this code:

```
success: function (data) {
  $("#modal-book .modal-content").html(data.html_form);
}
```

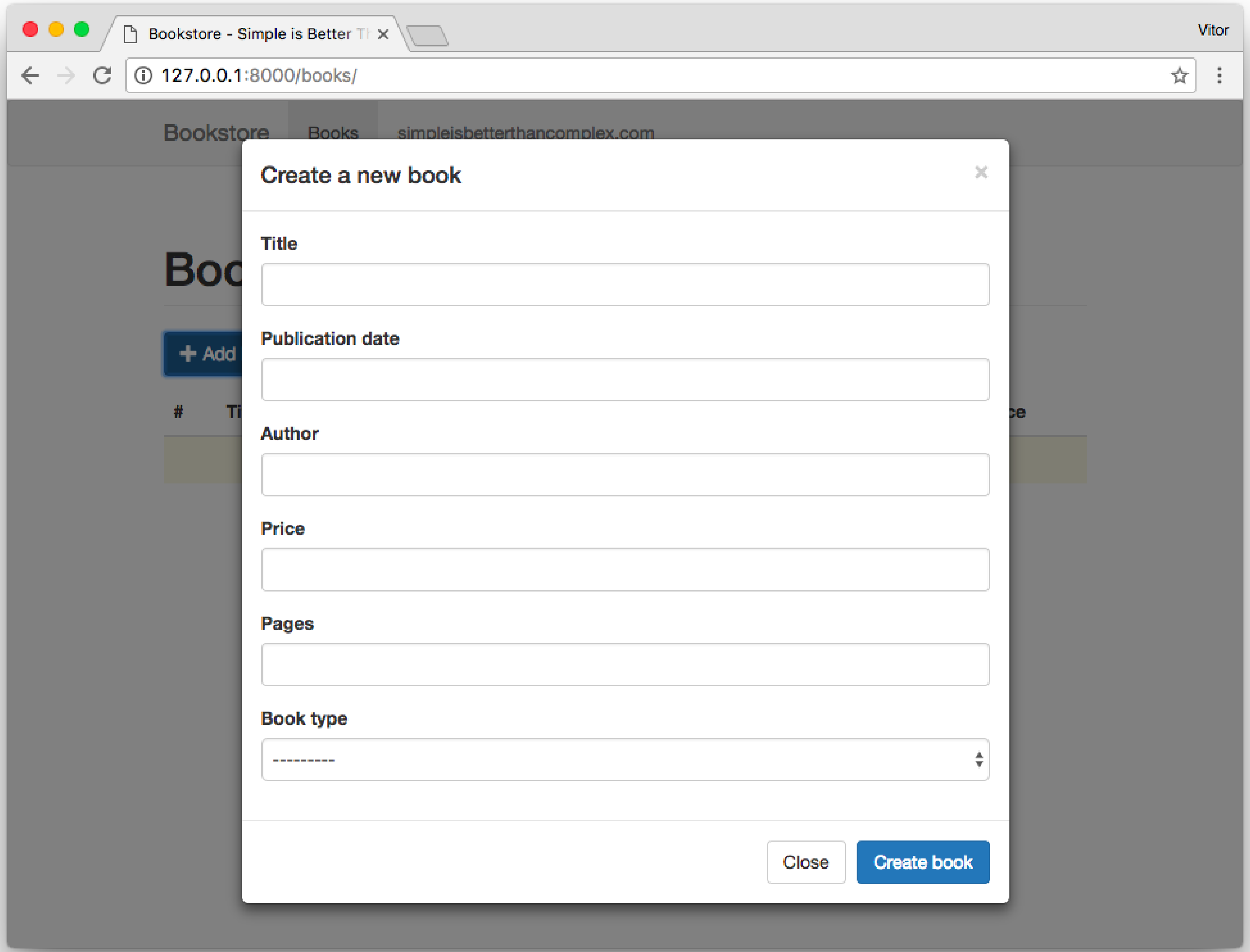
(This will render the partial form defined in the **partial_book_create.html** template.)

Let’s have a look on what we have so far:


Bookstore - Simple is Better Th
Vitor
127.0.0.1:8000/books/
Bookstore
Books
simpleisbetterthancomplex.com
Books
+ Add book

#	Title	Author	Type	Publication date	Pages	Price
			No book			

Then when the user clicks the button:


Bookstore - Simple is Better Th
Vitor
127.0.0.1:8000/books/
Bookstore
Books
simpleisbetterthancomplex.com
Boo
+ Add
#
Ti
ce

Create a new book

Title

Publication date

Author

Price

Pages

Book type

Great stuff. The book form is being rendered asynchronously. But it is not doing much at the moment. Good news is that the structure is ready, now it is a matter of playing with the data.

Let's implement now the form submission handling.

First let's improve the **book_create** view function:

views.py

```
def book_create(request):
    data = dict()

    if request.method == 'POST':
        form = BookForm(request.POST)
        if form.is_valid():
            form.save()
            data['form_is_valid'] = True
        else:
            data['form_is_valid'] = False
    else:
        form = BookForm()

    context = {'form': form}
    data['html_form'] = render_to_string('books/includes/partial_book_create.html',
        context,
        request=request
    )
    return JsonResponse(data)
```

partial_book_create.html

```
{% load widget_tweaks %}

<form method="post" action="{% url 'book_create' %}" class="js-book-create-form">
  <!-- FORM CONTENT SUPPRESSED FOR BREVITY'S SAKE -->
</form>
```

I added the **action** attribute to tell the browser to where it should send the submission and the class **js-book-create-form** for us to use in the JavaScript side, hooking on the form submit event.

books.js

```
$("#modal-book").on("submit", ".js-book-create-form", function () {
  ...
});
```

The way we are listening to the **submit** event is a little bit different from what we have implemented before. That's because the element with class **.js-book-create-form** didn't exist on the initial page load of the **book_list.html** template. So we can't register a listener to an element that doesn't exists.

A work around is to register the listener to an element that will always exist in the page context. The **#modal-book** is the closest element. It is a little bit more complex what happen, but long story short, the HTML events propagate to the parents elements until it reaches the end of the document.

Hooking to the **body** element would have the same effect, but it would be slightly worst, because it would have to travel through several HTML elements before reaching it. So always pick the closest one.

Now the actual function:

books.js

```
$("#modal-book").on("submit", ".js-book-create-form", function () {
  var form = $(this);
  $.ajax({
    url: form.attr("action"),
    data: form.serialize(),
    type: form.attr("method"),
    dataType: 'json',
    success: function (data) {
      if (data.form_is_valid) {
        alert("Book created!");  // <-- This is just a placeholder for now for testing
      }
      else {
        $("#modal-book .modal-content").html(data.html_form);
      }
    }
  });
  return false;
});
```

A very important detail here: in the end of the function we are returning **false**. That's because we are capturing the form submission event. So to avoid the browser to perform a full HTTP POST to the server, we cancel the default behavior returning false in the function.

So, what we are doing here:

```
var form = $(this);
```

In this context, `this` refers to the element with class `.js-book-create-form`. Which is the element that fired the submit event. So when we select `$(this)` we are selecting the actual form.

```
url: form.attr("action"),
```

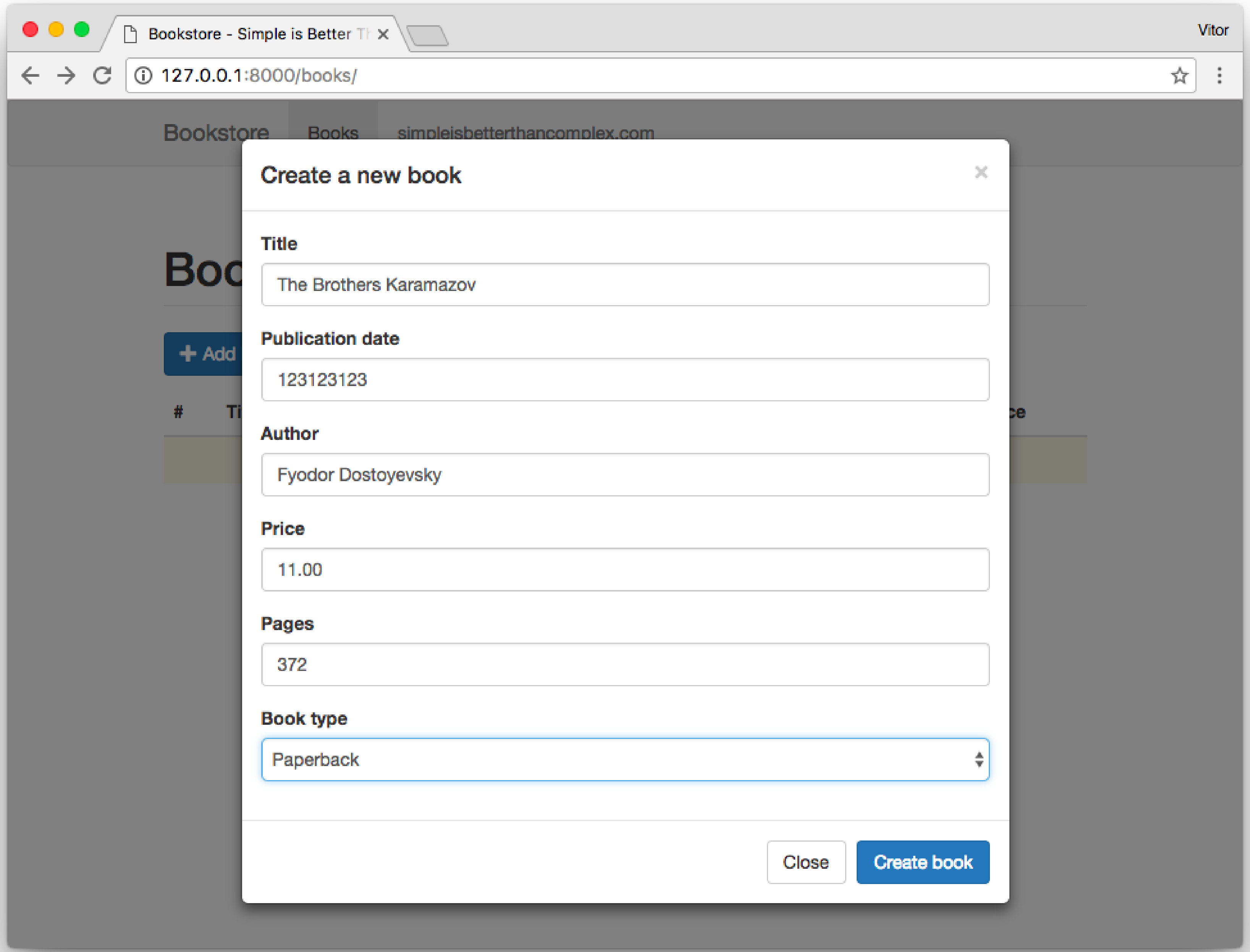
Now I'm using the form attributes to build the Ajax request. The **action** here refers to the **action** attribute in the **form**, which translates to `/books/create/`.

```
data: form.serialize(),
```

As the name suggests, we are serializing all the data from the form, and posting it to the server. The rest follows the same concepts as I explained before.

Before we move on, let's have a look on what we have so far.

The user fills the data:


Bookstore - Simple is Better Th
Vitor
127.0.0.1:8000/books/
Bookstore
Books
simpleisbetterthancomplex.com
Boo
+ Add
#
Ti
ce

Create a new book

Title

The Brothers Karamazov

Publication date

123123123

Author

Fyodor Dostoyevsky

Price

11.00

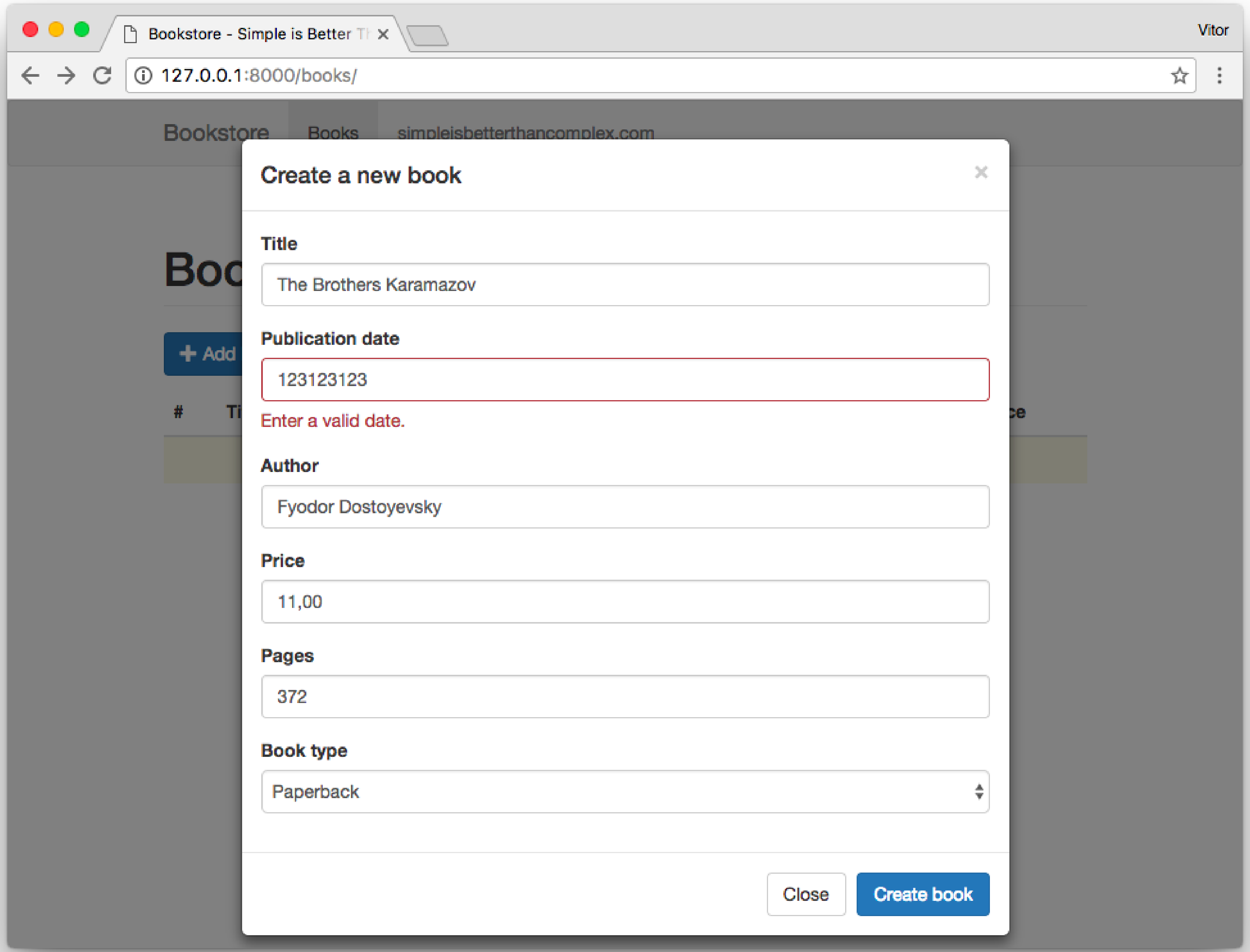
Pages

372

Book type

Paperback

The user clicks on the **Create book** button:



Create a new book

Title

The Brothers Karamazov

Publication date

123123123

Enter a valid date.

Author

Fyodor Dostoyevsky

Price

11,00

Pages

372

Book type

Paperback

The data was invalid. No hard refresh no anything. Just this tiny part changed with the validation. This is what happened:

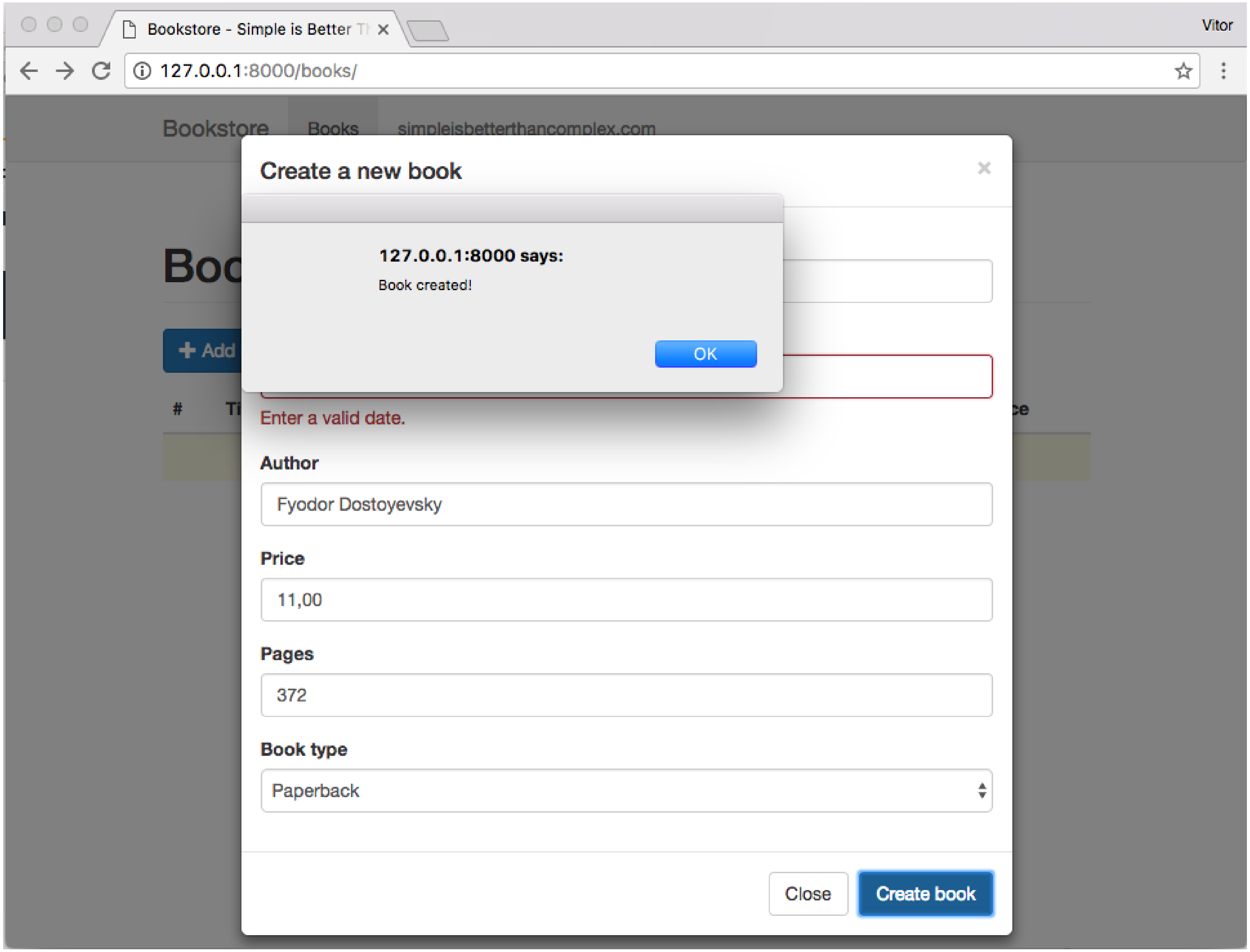
1. The form was submitted via Ajax
2. The view function processed the form
3. The form data was invalid
4. The view function rendered the invalid stated to the `data['html_form']`, using the **render_to_string**
5. The Ajax request returned to the JavaScript function
6. The Ajax **success** callback was executed, replacing the contents of the modal with the new `data['html_form']`

Please note that the Ajax **success** callback:

```
$.ajax({
  // ...
  success: function (data) {
    // ...
  }
});
```

Refers to the status of the **HTTP Request**, which has nothing to do with the status of your form, or whether the form was successfully processed or not. It only means that the **HTTP Request** returned a status **200** for example.

Let's fix the **publication date** value and submit the form again:



Create a new book

127.0.0.1:8000 says:

Book created!

Enter a valid date.

Author: Fyodor Dostoyevsky

Price: 11,00

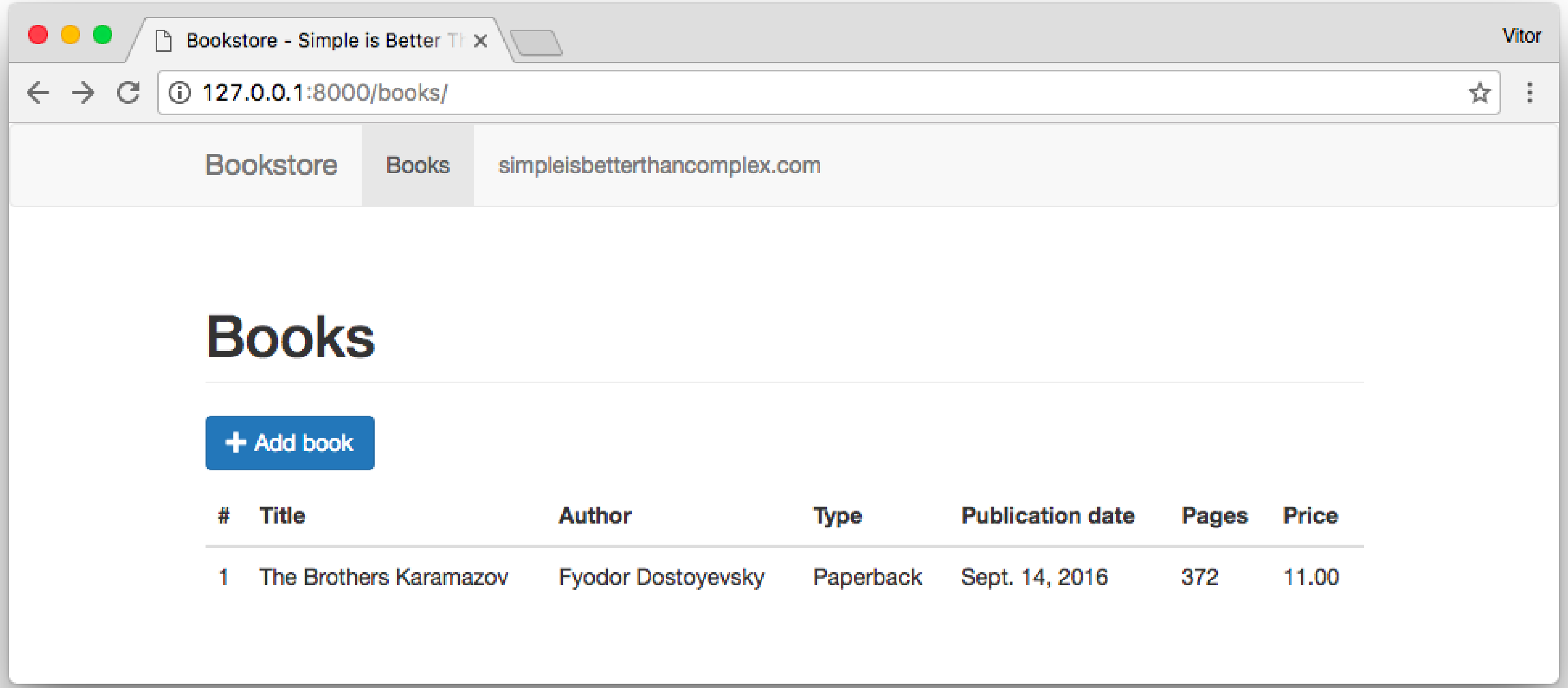
Pages: 372

Book type: Paperback

There we go, the **alert** tells us that the form was successfully processed and hopefully it was created in the database.

```
success: function (data) {
  if (data.form_is_valid) {
    alert("Book created!");  // <-- This line was executed! Means success
  }
  else {
    $("#modal-book .modal-content").html(data.html_form);
  }
}
```

It is not 100% what we want, but we are getting close. Let's refresh the screen and see if the new book shows in the table:


Bookstore - Simple is Better Th
Vitor
127.0.0.1:8000/books/
Bookstore
Books
simpleisbetterthancomplex.com
Books
+ Add book

#	Title	Author	Type	Publication date	Pages	Price
1	The Brothers Karamazov	Fyodor Dostoyevsky	Paperback	Sept. 14, 2016	372	11.00

Great. We are getting there.

What we want to do now: after the success form processing, we want to close the bootstrap modal *and* update the table with the newly created book. For that matter we will extract the body of the table to a external partial template, and we will return the new table body in the Ajax success callback.

Watch that:

book_list.html

```
<table class="table" id="book-table">
  <thead>
    <tr>
      <th>#</th>
      <th>Title</th>
      <th>Author</th>
      <th>Type</th>
      <th>Publication date</th>
      <th>Pages</th>
      <th>Price</th>
    </tr>
  </thead>
  <tbody>
    {% include 'books/includes/partial_book_list.html' %}
  </tbody>
</table>
```

partial_book_list.html

```
{% for book in books %}
  <tr>
    <td>{{ book.id }}</td>
    <td>{{ book.title }}</td>
    <td>{{ book.author }}</td>
    <td>{{ book.get_book_type_display }}</td>
    <td>{{ book.publication_date }}</td>
    <td>{{ book.pages }}</td>
    <td>{{ book.price }}</td>
  </tr>
{% empty %}
  <tr>
    <td colspan="7" class="text-center bg-warning">No book</td>
  </tr>
{% endfor %}
```

Now we can reuse the **partial_book_list.html** snippet without repeating ourselves.

Next step: **book_create** view function.

views.py

```
def book_create(request):
    data = dict()

    if request.method == 'POST':
        form = BookForm(request.POST)
        if form.is_valid():
            form.save()
            data['form_is_valid'] = True
            books = Book.objects.all()
            data['html_book_list'] = render_to_string('books/includes/partial_book_list.html', {
                'books': books
            })
        else:
            data['form_is_valid'] = False
    else:
        form = BookForm()

    context = {'form': form}
    data['html_form'] = render_to_string('books/includes/partial_book_create.html',
        context,
        request=request
    )
    return JsonResponse(data)
```

A proper success handler in the JavaScript side:

books.js

```
  $("#modal-book").on("submit", ".js-book-create-form", function () {
    var form = $(this);
    $.ajax({
      url: form.attr("action"),
      data: form.serialize(),
      type: form.attr("method"),
      dataType: 'json',
      success: function (data) {
        if (data.form_is_valid) {
          $("#book-table tbody").html(data.html_book_list);  // <-- Replace the table body
          $("#modal-book").modal("hide");  // <-- Close the modal
        }
        else {
          $("#modal-book .modal-content").html(data.html_form);
        }
      }
    });
    return false;
  });
```

Sweet. It’s working!

Edit Book

As you can expect, this will be very similar to what we did on the Create Book section. Except we will need to pass the ID of the book we want to edit. The rest should be somewhat the same. We will be reusing several parts of the code.

urls.py:

```
from django.conf.urls import url, include
from mysite.books import views

urlpatterns = [
    url(r'^books/$', views.book_list, name='book_list'),
    url(r'^books/create/$', views.book_create, name='book_create'),
    url(r'^books/(?P<pk>\d+)/update/$', views.book_update, name='book_update'),
]
```

Now we refactor the **book_create** view to reuse its code in the **book_update** view:

views.py

```
from django.shortcuts import render, get_object_or_404
from django.http import JsonResponse
from django.template.loader import render_to_string

from .models import Book
from .forms import BookForm


def save_book_form(request, form, template_name):
    data = dict()
    if request.method == 'POST':
        if form.is_valid():
            form.save()
            data['form_is_valid'] = True
            books = Book.objects.all()
            data['html_book_list'] = render_to_string('books/includes/partial_book_list.html', {
                'books': books
            })
        else:
            data['form_is_valid'] = False
    context = {'form': form}
    data['html_form'] = render_to_string(template_name, context, request=request)
    return JsonResponse(data)


def book_create(request):
    if request.method == 'POST':
        form = BookForm(request.POST)
    else:
        form = BookForm()
    return save_book_form(request, form, 'books/includes/partial_book_create.html')


def book_update(request, pk):
    book = get_object_or_404(Book, pk=pk)
    if request.method == 'POST':
        form = BookForm(request.POST, instance=book)
    else:
        form = BookForm(instance=book)
    return save_book_form(request, form, 'books/includes/partial_book_update.html')
```

Basically the view functions **book_create** and **book_update** are responsible for receiving the request, preparing the form instance and passing it to the **save_book_form**, along with the name of the template to use in the rendering process.

Next step is to create the **partial_book_update.html** template. Similar to what we did with the view functions, we will also refactor the **partial_book_create.html** to reuse some of the code.

partial_book_form.html

```
{% load widget_tweaks %}

{% for field in form %}
  <div class="form-group{% if field.errors %} has-error{% endif %}">
    <label for="{{ field.id_for_label }}">{{ field.label }}</label>
    {% render_field field class="form-control" %}
    {% for error in field.errors %}
      <p class="help-block">{{ error }}</p>
    {% endfor %}
  </div>
{% endfor %}
```

partial_book_create.html

```
<form method="post" action="{% url 'book_create' %}" class="js-book-create-form">
  {% csrf_token %}
  <div class="modal-header">
    <button type="button" class="close" data-dismiss="modal" aria-label="Close">
      <span aria-hidden="true">&times;</span>
    </button>
    <h4 class="modal-title">Create a new book</h4>
  </div>
  <div class="modal-body">
    {% include 'books/includes/partial_book_form.html' %}
  </div>
  <div class="modal-footer">
    <button type="button" class="btn btn-default" data-dismiss="modal">Close</button>
    <button type="submit" class="btn btn-primary">Create book</button>
  </div>
</form>
```

partial_book_update.html

```
<form method="post" action="{% url 'book_update' form.instance.pk %}" class="js-book-update-form">
  {% csrf_token %}
  <div class="modal-header">
    <button type="button" class="close" data-dismiss="modal" aria-label="Close">
      <span aria-hidden="true">&times;</span>
    </button>
    <h4 class="modal-title">Update book</h4>
  </div>
  <div class="modal-body">
    {% include 'books/includes/partial_book_form.html' %}
  </div>
  <div class="modal-footer">
    <button type="button" class="btn btn-default" data-dismiss="modal">Close</button>
    <button type="submit" class="btn btn-primary">Update book</button>
  </div>
</form>
```

This is good enough. Now we gotta add an edit button to trigger the action.

partial_book_list.html

```
{% for book in books %}
  <tr>
    <td>{{ book.id }}</td>
    <td>{{ book.title }}</td>
    <td>{{ book.author }}</td>
    <td>{{ book.get_book_type_display }}</td>
    <td>{{ book.publication_date }}</td>
    <td>{{ book.pages }}</td>
    <td>{{ book.price }}</td>
    <td>
      <button type="button"
              class="btn btn-warning btn-sm js-update-book"
              data-url="{% url 'book_update' book.id %}">
        <span class="glyphicon glyphicon-pencil"></span> Edit
      </button>
    </td>
  </tr>
{% empty %}
```

```
  <tr>
    <td colspan="8" class="text-center bg-warning">No book</td>
  </tr>
{% endfor %}
```

The class **js-update-book** will be used to start the edit process. Now note that I also added an extra HTML attribute named **data-url.** This is the URL that will be used to create the ajax request dynamically.

Take the time and refactor the **js-create-book** button to also use the **data-url** strategy, so we can extract the hard-coded URL from the Ajax request.

book_list.html

```
{% extends 'base.html' %}

{% block content %}
  <h1 class="page-header">Books</h1>

  <p>
    <button type="button"
            class="btn btn-primary js-create-book"
            data-url="{% url 'book_create' %}">
      <span class="glyphicon glyphicon-plus"></span>
      New book
    </button>
  </p>

  <!-- REST OF THE PAGE... -->

{% endblock %}
```

books.js

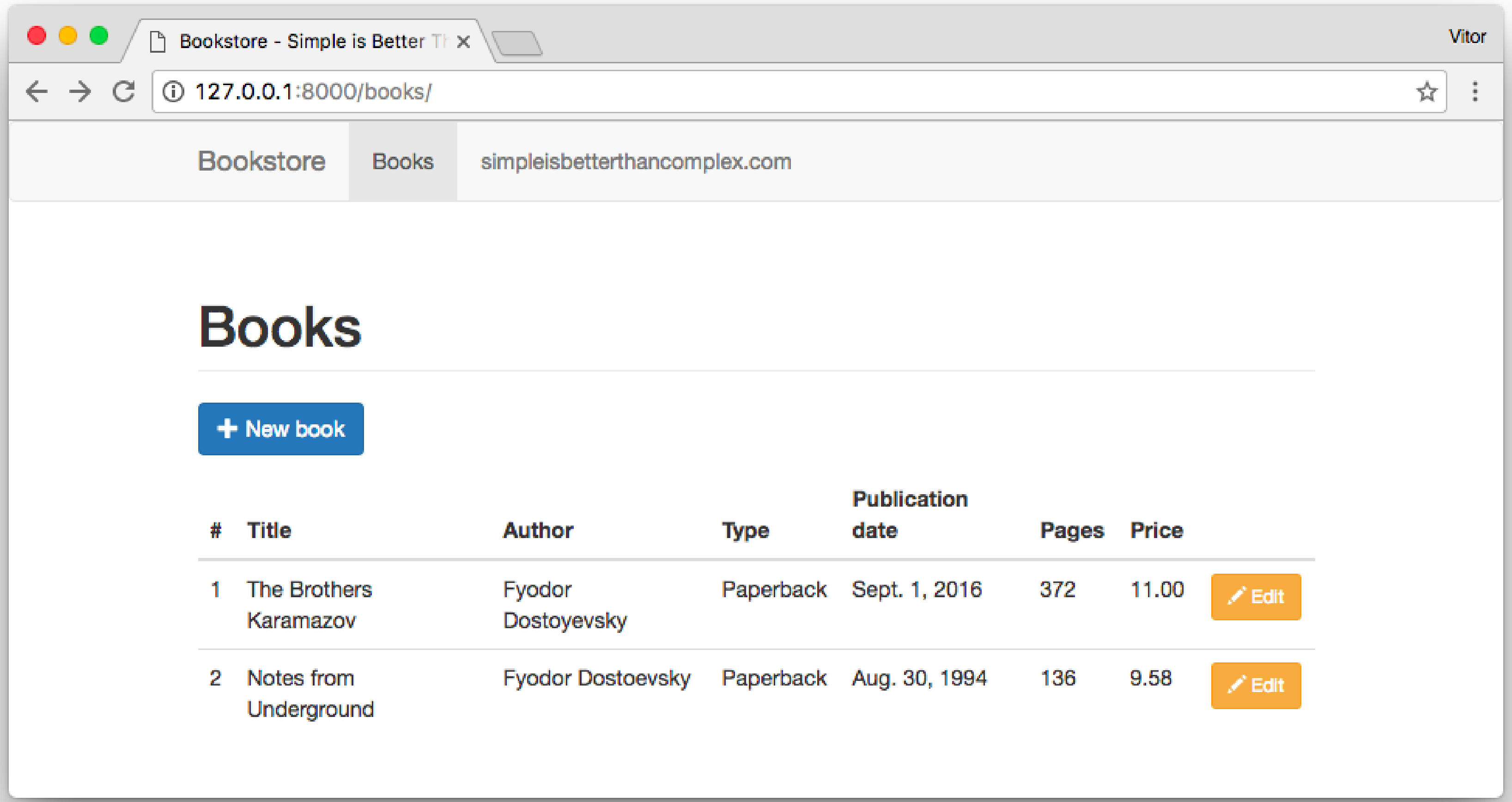
```
$(".js-create-book").click(function () {
  var btn = $(this);  // <-- HERE
  $.ajax({
    url: btn.attr("data-url"),  // <-- AND HERE
    type: 'get',
    dataType: 'json',
    beforeSend: function () {
      $("#modal-book").modal("show");
    },
    success: function (data) {
      $("#modal-book .modal-content").html(data.html_form);
    }
  });
});
```

Next step is to create the edit functions. The thing is, they are pretty much the same as the create. So, basically what we want to do is to extract the anonymous functions that we are using, and reuse them in the edit buttons and forms. Check it out:

books.js

```
$(function () {

  /* Functions */

  var loadForm = function () {
    var btn = $(this);
    $.ajax({
      url: btn.attr("data-url"),
      type: 'get',
      dataType: 'json',
      beforeSend: function () {
        $("#modal-book").modal("show");
      },
      success: function (data) {
        $("#modal-book .modal-content").html(data.html_form);
      }
    });
  };

  var saveForm = function () {
    var form = $(this);
    $.ajax({
      url: form.attr("action"),
      data: form.serialize(),
      type: form.attr("method"),
      dataType: 'json',
      success: function (data) {
        if (data.form_is_valid) {
          $("#book-table tbody").html(data.html_book_list);
          $("#modal-book").modal("hide");
        }
        else {
          $("#modal-book .modal-content").html(data.html_form);
        }
      }
    });
    return false;
  };


  /* Binding */

  // Create book
  $(".js-create-book").click(loadForm);
  $("#modal-book").on("submit", ".js-book-create-form", saveForm);

  // Update book
  $("#book-table").on("click", ".js-update-book", loadForm);
  $("#modal-book").on("submit", ".js-book-update-form", saveForm);

});
```

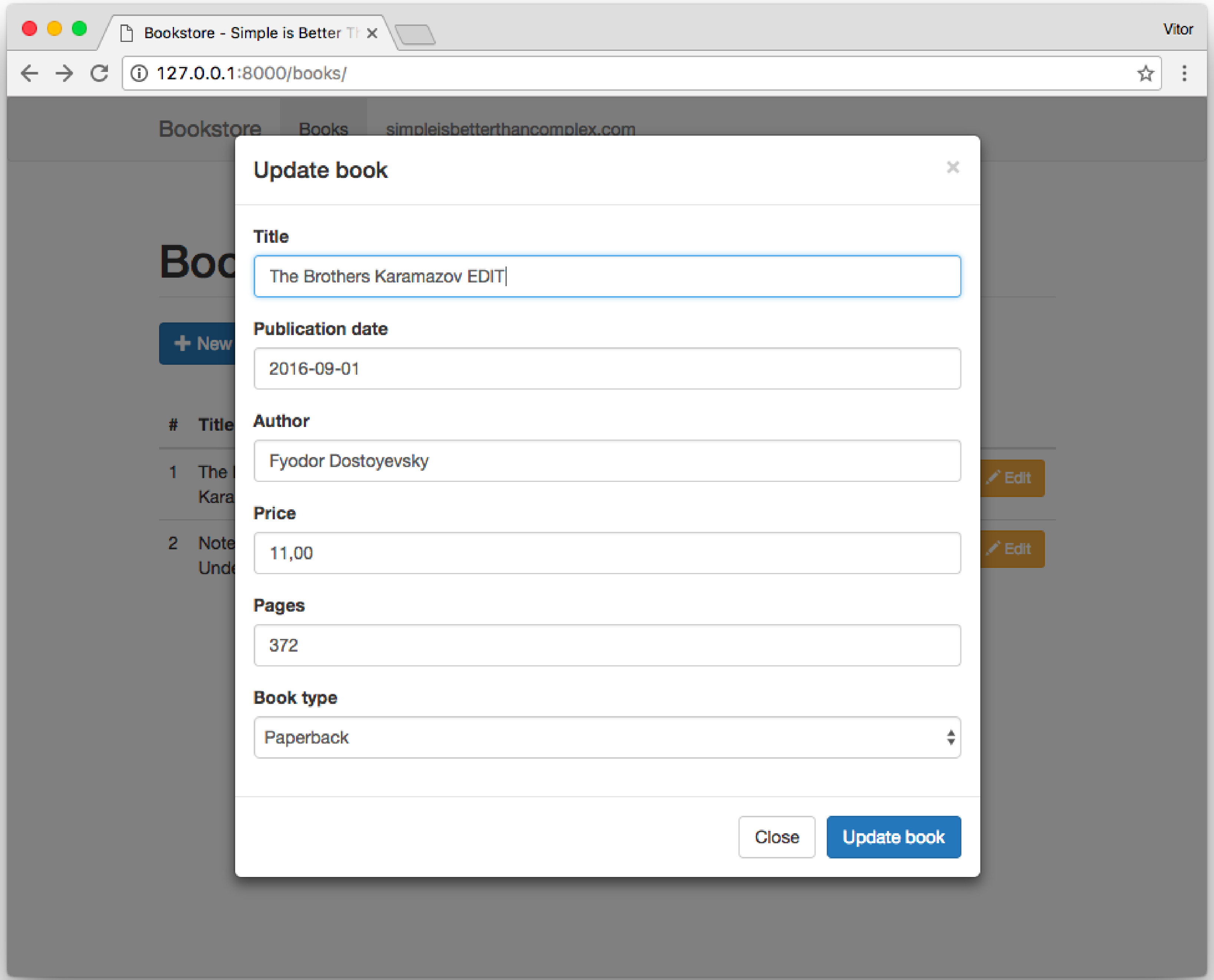
Let’s have a look on what we have so far.


Bookstore - Simple is Better Th
Vitor
127.0.0.1:8000/books/
Bookstore
Books
simpleisbetterthancomplex.com
+ New book
Edit
Edit

Books

#	Title	Author	Type	Publication date	Pages	Price	
1	The Brothers Karamazov	Fyodor Dostoyevsky	Paperback	Sept. 1, 2016	372	11.00	Edit
2	Notes from Underground	Fyodor Dostoevsky	Paperback	Aug. 30, 1994	136	9.58	Edit

The user clicks in the edit button.


Bookstore - Simple is Better Tł
Vitor
127.0.0.1:8000/books/
Bookstore
Books
simpleisbetterthancomplex.com
Boo
New
Edit
Edit

Update book

Title

The Brothers Karamazov EDIT

Publication date

2016-09-01

Author

Fyodor Dostoyevsky

Price

11,00

Pages

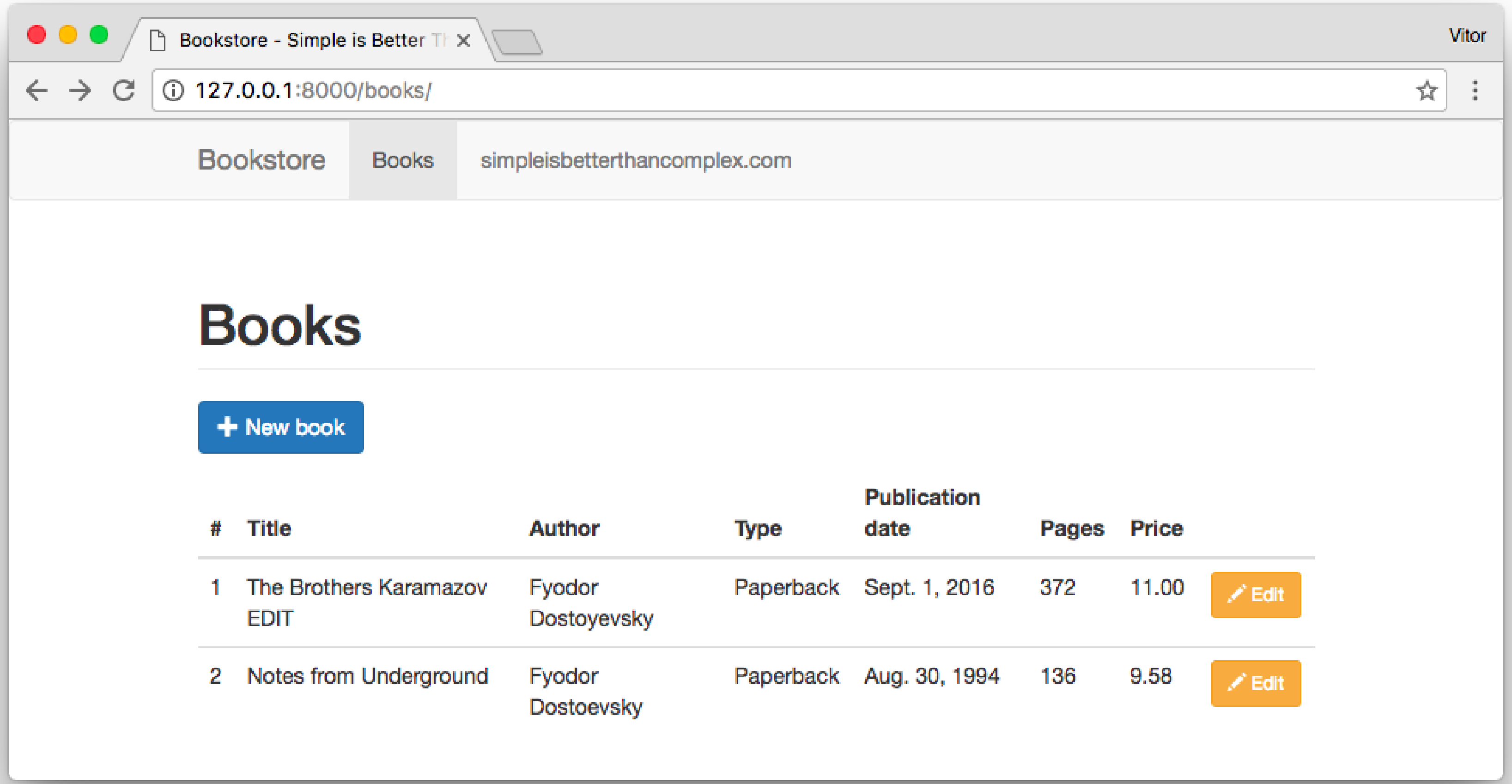
372

Book type

Paperback

Close Update book

Changes some data like the title of the book and hit the **Update book** button:


Bookstore - Simple is Better Th
Vitor
127.0.0.1:8000/books/
Bookstore
Books
simpleisbetterthancomplex.com
+ New book
Edit
Edit

Books

#	Title	Author	Type	Publication date	Pages	Price	
1	The Brothers Karamazov EDIT	Fyodor Dostoyevsky	Paperback	Sept. 1, 2016	372	11.00	Edit
2	Notes from Underground	Fyodor Dostoevsky	Paperback	Aug. 30, 1994	136	9.58	Edit

Cool! Now just the delete and we are done.

Delete Book

urls.py:

```
from django.conf.urls import url, include
from mysite.books import views

urlpatterns = [
    url(r'^books/$', views.book_list, name='book_list'),
    url(r'^books/create/$', views.book_create, name='book_create'),
    url(r'^books/(?P<pk>\d+)/update/$', views.book_update, name='book_update'),
    url(r'^books/(?P<pk>\d+)/delete/$', views.book_delete, name='book_delete'),
]
```

views.py

```
def book_delete(request, pk):
    book = get_object_or_404(Book, pk=pk)
    data = dict()
    if request.method == 'POST':
        book.delete()
        data['form_is_valid'] = True  # This is just to play along with the existing code
        books = Book.objects.all()
        data['html_book_list'] = render_to_string('books/includes/partial_book_list.html', {
            'books': books
        })
    else:
        context = {'book': book}
        data['html_form'] = render_to_string('books/includes/partial_book_delete.html',
            context,
            request=request,
        )
    return JsonResponse(data)
```

partial_book_delete.html

```
<form method="post" action="{% url 'book_delete' book.id %}" class="js-book-delete-form">
  {% csrf_token %}
  <div class="modal-header">
    <button type="button" class="close" data-dismiss="modal" aria-label="Close">
      <span aria-hidden="true">&times;</span>
    </button>
    <h4 class="modal-title">Confirm book deletion</h4>
  </div>
  <div class="modal-body">
    <p class="lead">Are you sure you want to delete the book <strong>{{ book.title }}</strong>?</p>
  </div>
  <div class="modal-footer">
    <button type="button" class="btn btn-default" data-dismiss="modal">Close</button>
    <button type="submit" class="btn btn-danger">Delete book</button>
  </div>
</form>
```

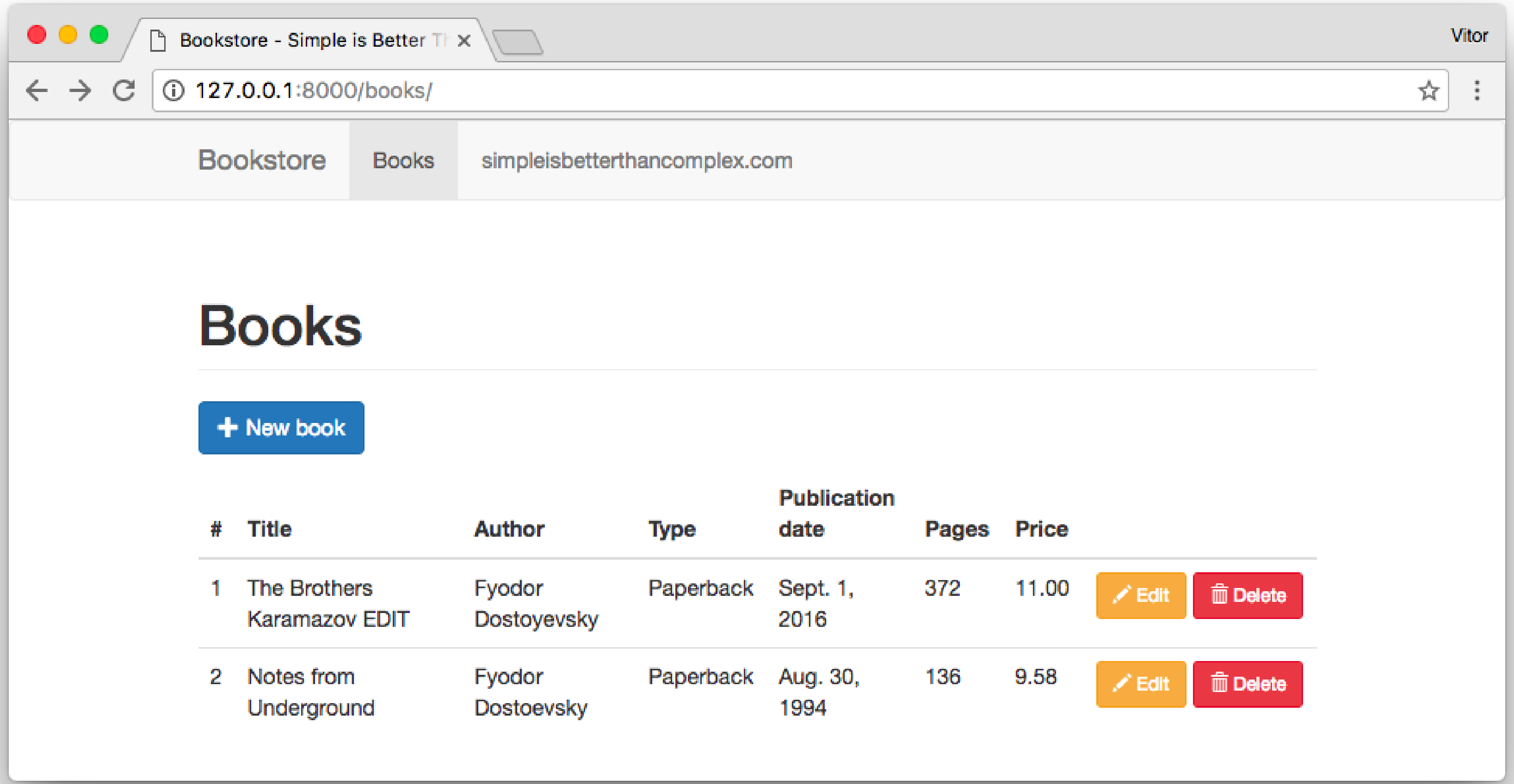
partial_book_list.html

```
{% for book in books %}
  <tr>
    <td>{{ book.id }}</td>
    <td>{{ book.title }}</td>
    <td>{{ book.author }}</td>
    <td>{{ book.get_book_type_display }}</td>
    <td>{{ book.publication_date }}</td>
    <td>{{ book.pages }}</td>
    <td>{{ book.price }}</td>
    <td>
      <button type="button"
              class="btn btn-warning btn-sm js-update-book"
              data-url="{% url 'book_update' book.id %}">
        <span class="glyphicon glyphicon-pencil"></span> Edit
      </button>
      <button type="button"
              class="btn btn-danger btn-sm js-delete-book"
              data-url="{% url 'book_delete' book.id %}">
        <span class="glyphicon glyphicon-trash"></span> Delete
      </button>
    </td>
  </tr>
{% empty %}
  <tr>
    <td colspan="8" class="text-center bg-warning">No book</td>
  </tr>
{% endfor %}
```

books.js

```
// Delete book
$("#book-table").on("click", ".js-delete-book", loadForm);
$("#modal-book").on("submit", ".js-book-delete-form", saveForm);
```

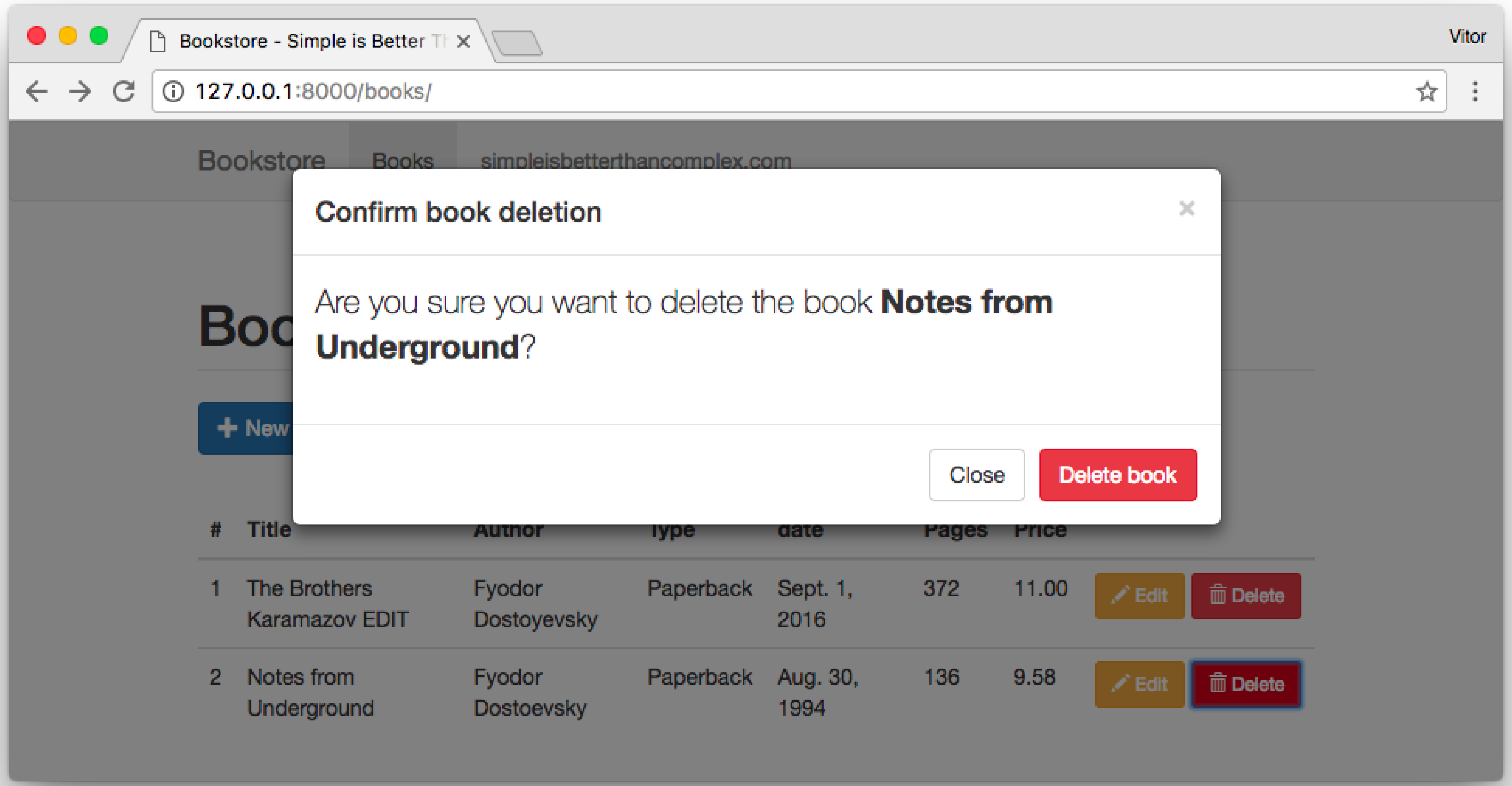
And the result will be:


Bookstore - Simple is Better Th
Vitor
127.0.0.1:8000/books/
Bookstore
Books
simpleisbetterthancomplex.com
+ New book
Edit
Delete

Books

#	Title	Author	Type	Publication date	Pages	Price	
1	The Brothers Karamazov EDIT	Fyodor Dostoyevsky	Paperback	Sept. 1, 2016	372	11.00	Edit Delete
2	Notes from Underground	Fyodor Dostoevsky	Paperback	Aug. 30, 1994	136	9.58	Edit Delete

The user clicks in a Delete button:



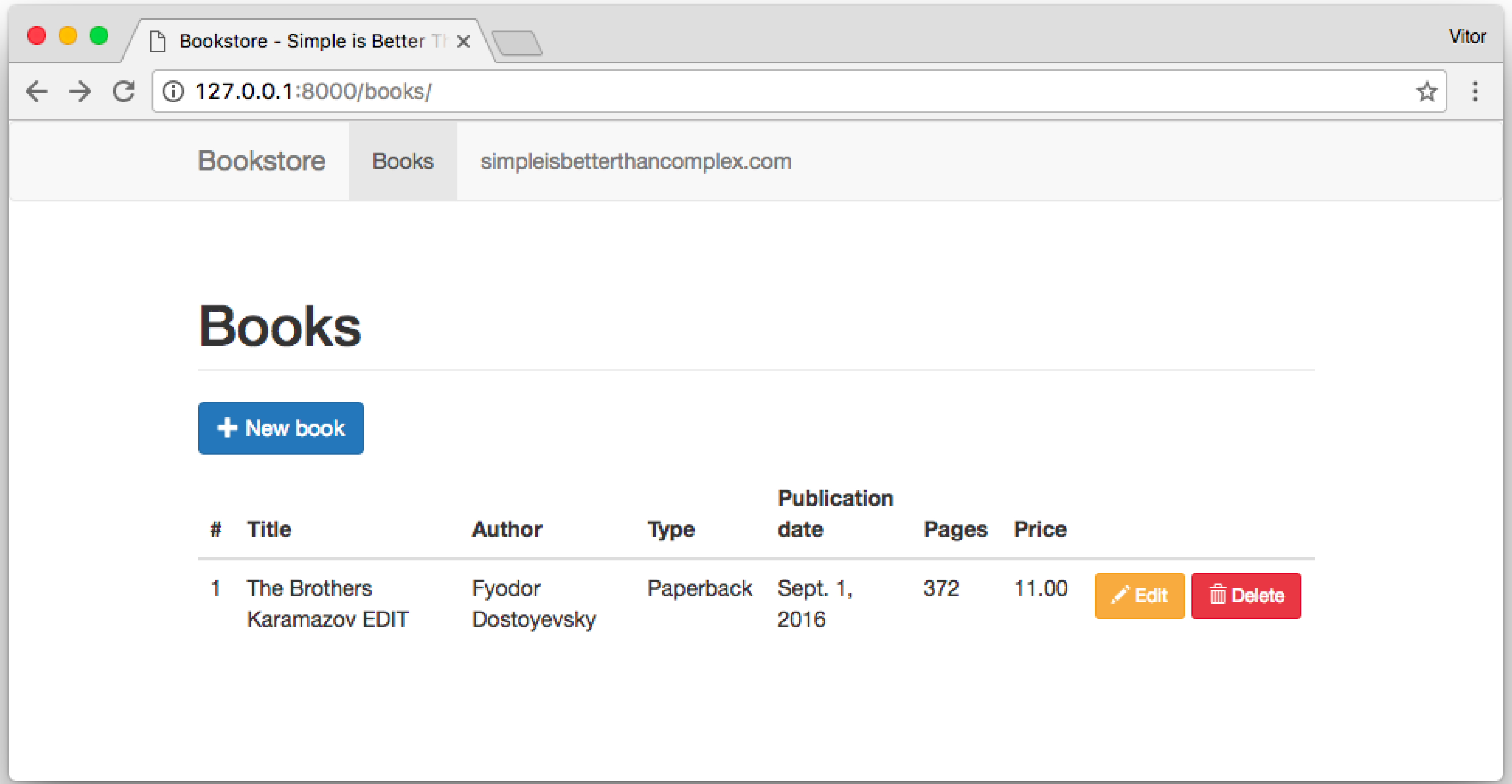
Confirm book deletion

Are you sure you want to delete the book **Notes from Underground**?

Bookstore - Simple is Better Th
Vitor
127.0.0.1:8000/books/
Bookstore
Books
simpleisbetterthancomplex.com
×
Close
Delete book
New

#	Title	Author	Type	date	Pages	Price	
1	The Brothers Karamazov EDIT	Fyodor Dostoyevsky	Paperback	Sept. 1, 2016	372	11.00	Edit Delete
2	Notes from Underground	Fyodor Dostoevsky	Paperback	Aug. 30, 1994	136	9.58	Edit Delete

The user confirm the deletion, and the table in the background is refreshed:



#	Title	Author	Type	Publication date	Pages	Price	
1	The Brothers Karamazov EDIT	Fyodor Dostoyevsky	Paperback	Sept. 1, 2016	372	11.00	Edit Delete

Conclusions

I decided to use function-based views in this example because they are easier to read and use less configuration-magic that could distract those who are learning Django.

I tried to bring as much detail as I could and discuss about some of the code design decisions. If anything is not clear to you, or you want to suggest some improvements, feel free to leave a comment! I would love to hear your thoughts.

The code is available on GitHub: github.com/sibtc/simple-ajax-crud, so you can try it locally. It's very straightforward to get it running.

Related Posts

How to Create Infinite Scroll With Django

How to Crop Images in a Django Application

[Django Multiple Files Upload Using Ajax](https://simpleisbetterthancomplex.com)

django ajax crud jquery

Like 26 12

Share this post

71 Comments Simple is Better Than Complex 1 Login

Recommend 10 Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Jan Wilmar • 10 months ago

This is amazing! Thank you.

3 • Reply • Share ›

Vitor Freitas Mod → Jan Wilmar • 10 months ago

Thanks Jan!

• Reply • Share ›

iMitwe • 10 months ago

Thanks very much. it's awesome. and neat.

2 • Reply • Share ›

Vitor Freitas Mod → iMitwe • 10 months ago

Thanks!!

• Reply • Share ›

Oskar Gmerek • 2 months ago

I do everything step by step from your tutorial but when i click on New book button I dont getting form in modal, modal is showing totally empty. In firebug I can see completed xhr get request. In this xhr request I can see me form. But without firebug I can see only empty modal (not empty form - no form and no other data is in this modal) What I can do wrong? I using python 3.5.2, django 1.11.2, jquery 3.2.1, bootstrap 3.3.7

Please help me ;)

1 • Reply • Share ›

Marcelo Grebois • 9 months ago

Hi Victor, first of I would like to thank you for this awesome tutorial, its amazing.

Now I have a couple of newbie questions, hope they don't so very stupid:

1) I know you use json to interact with the modals, is there any advantages using this approach against the standard class-views and then loading the full form ?

I was already able to set up a very similar CRUD using class views and the views.py look far more simpler, although the templates might not.

2) I tried to modify the form to have autocomplete in some fields, but the widget does not get loaded, although if I change the template to render as {{ form|crispy }}, then it shows just fine, any idea what I might be missing?

```
class BookForm(forms.ModelForm):
    class Meta:
        model = Book
        fields = ('__all__')
        exclude = ('user',)
        widgets = {
            'city': autocomplete.ModelSelect2(url='city')
        }
```

3) When I access /books, book_list.html get render perfectly:

```
def book_list(request):
    books = book.objects.all().filter(user=request.user)
    return render(request, 'books/book_list.html', {'books': books})
```

but as soon as I try to include this on the main page, it does not get render:

```
{% include 'books/book_list.html' %}
```

If I add a book the list appear, but then as soon as I refresh its gone again.

1 ^ | ˅ • Reply • Share ›

Vitor Freitas Mod → Marcelo Grebois • 9 months ago

Hey Marcelo!
Thanks for your comment :-)

1) The main reason to work with the Json responses is to have more control over the data we return to the client and control the behavior inside the Ajax function, displaying the form inside the modal with error state, checking either if the form was valid or not, refreshing the table with the new entry and so on. This could also be achieved with normal HttpResponse or even simply using `return render(request, 'template_name.html')` etc. But then the return would simply be a string, so inside the Ajax function, a possible handler would be:

```
success: function (data) {
    $("#book-table tbody").html(data);  // instead of data.html_book_list
    $("#modal-book").modal("hide");
}
```

But then again, it would be tricky to test if the form was properly processed.

Now you can still use a class-based views returning JsonResponse. So you would have both: view classes + json.

I agree with you that we can make the code smaller using class-based views. But usually I try to avoid using them in tutorials here in the blog. Usually function-based views are more explicit so it's more clear to understand what is going

see more

1 ^ | ˅ • Reply • Share ›

Marcelo Grebois ➦ Vitor Freitas • 9 months ago

Hey man, many thanks for taking the time and getting back to me this soon. I wasn't able to make the autocomplete widget work by assigning it within the Meta class, so I move forward and finished it with class-views.

Now for the last issue I really cannot find whats wrong, I removed the user filter just to make sure but still, this url will show me the list of 'trips' in my case:

/trips -> trip_list.html

```
{% extends 'base.html' %}

{% block content %}
      {% include 'trips/trip_list_base.html' %}
{% endblock %}
```

but this url will not show any result:

/ -> home.html

```
{% extends 'base.html' %}

{% block content %}
<div class="col-sm-5">
    <div class="row">
        <div class="col-sm-12">
            {% include 'trips/trip_list_base.html' %}
        </div>
    </div>
</div>
{% endblock %}
```

I uploaded a full example, maybe you could take a quick look: https://github.com/grebois/...
Many Thanks!

Reply • Share ›

Marcelo Grebois → Marcelo Grebois • 9 months ago

I was able to make it work :) wants rendering the right tags :/ Thank you so much for this articule.

Reply • Share ›

Vitor Freitas Mod → Marcelo Grebois • 9 months ago

Sorry I couldn't check this earlier!
I'm glad to hear to figured out and made it work :-)

Reply • Share ›

Ruben Musalia • 10 months ago

Definitely trying this out tomorrow morning God willing as an exercise .Thank you.

1 • Reply • Share ›

Vitor Freitas Mod → Ruben Musalia • 10 months ago

Nice one!
Let me know if you need any help :-)

Reply • Share ›

Ruben Musalia → Vitor Freitas • 10 months ago

I'm getting this AttibuteError.

see more

^ | v • Reply • Share ›

Vitor Freitas Mod ➦ Ruben Musalia • 10 months ago

Hi Ruben,

This error happened because you are using Django 1.9

I will fix the article to be compatible with earlier versions as well.

Basically you need to change the parameters in the **render_to_string** function:

data['html_form'] = render_to_string('books/includes/partial_book_create.html', context, request=request)

Basically everywhere you are using **render_to_string** and you see a **request** parameter, add it as a keyword argument like this **request=request**.

It is happening because it was changed in the 1.8 version, and in the 1.10 they dropped the backwards compatibility.

^ | v • Reply • Share ›

Karan Narvekar • 9 days ago

Hi can i have the same to work with mysql database ? are you planning to have a similar stuff to use with mysql database

• Reply • Share ›

Karan Narvekar • 15 days ago

Hello. Can you show how to connect Django with mysql. Do you have any post for it. This article was very helpful :D

• Reply • Share ›

Pappy • a month ago

Hi Vitor, thank you so much for this awesome tutorial. I have only created the book_list and book_create functions which I only want to show me the partial_create_book without saving or any further action. However, whenever I browse 127.0.0.1:8000/books, the form pops up without pressing the add button and this button doesn't do any anything.

I would really appreciate it, if you help me to figure out this as I am going to continue the rest of this awesome tutorial.

Thanks,

• Reply • Share ›

osama abuomar • a month ago

Thanks very much, very helpful

• Reply • Share ›

Rafael • 2 months ago

I'm trying create a CRUD using Ajax with pagination, but when I try add a new book, i get the exception empty in foreach of books, but when I refresh the page it works and show the new register.

def livro(request):
livros = Livro.objects.all().order_by('-id')
paginator = Paginator(livros, 5)

page = request.GET.get('page', 1)
try:
registers = paginator.page(page)
except PageNotAnInteger:
# If page is not an integer, deliver first page.

registers = paginator.page(1)
except EmptyPage:
# If page is out of range (e.g. 9999), deliver last page of results.
registers = paginator.page(paginator.num_pages)

return render(request, 'livros.html', {'registers': registers})

the rest still the same.

^ | ⌄ • Reply • Share ›

Colin Sturm • 2 months ago

Fantastic post Vitor! Thanks for creating this tutorial with screenshots. Just what I was looking for!

^ | ⌄ • Reply • Share ›

Ryan Manzer • 2 months ago

Thank you very much. This is a very helpful tutorial that I have implemented in a few different applications. One issue I have run into that might merit a tutorial or some follow up is how best to handle the messages framework when using AJAX and returning JsonResponses. I've added a custom function for loading my messages <div> but no matter how I structure my JS functions the request to my custom messages url is requested prior to the actual view function that will add the message. Secondly, I am having trouble clearing messages that have been viewed by the user without a page refresh. Any discussions of this would be very useful!

^ | ⌄ • Reply • Share ›

Pamungkas Sugihartoko • 2 months ago

Hii Vitor...

thanks for the amazing tutorial...
i'm new in python and django. i've read about pagination sample in your another post.
how to implement this tutorial with the pagination sample.?

many thanks vitor...

^ | ⌄ • Reply • Share ›

Mottaz Hejaze • 3 months ago

thank you very much man

Reply • Share ›

willian firmino • 3 months ago

I don't know what's happening with my code, when I try to insert a new person, it doesn't appear in the table,
When I refresh the page, this new record appears duplicated in the table. Any idea what's happening? Here my code:
https://gist.github.com/wil...

Reply • Share ›

Jeff Tilton • 5 months ago

Hi Vitor, thanks for the tutorial it is awesome and I have learned a lot struggling through it. Now that I am done, I wanted to add a many to one field with the user. In a previous project, I have done something like

```
user = request.user
form= BookForm(request.POST)
if form.is_valid():
     book = form.save(commit = False)
     book.user = user
```

however when I try to implement this in the save_book_view I get NOT NULL constraint failed books_book.user_id

I have also tried to set the form in book_create to user

```
user = request.user
if request.method == 'POST':
        form = BookForm(request.POST, instance = user)
```

It doesn't raise an error, but it doesn't save either. I am just learning Django, but this approach had worked on my first project, Any advice is appreciated. I have gone through several of your tutorials and I really appreciate your hard work! Thanks again

Reply • Share ›

eri vera • 5 months ago

me sale este error (en la consola) al ejcutar "python manage.py runserver" :

System check identified no issues (0 silenced).
April 02, 2017 - 23:59:45
Django version 1.10.3, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/

Quit the server with CTRL-BREAK.
Error: [WinError 10013] Intento de acceso a un socket no permitido por sus permisos de acceso

Reply • Share ›

Diego González Espinilla • 5 months ago

Hi Vitor.

How can I add a new button in the modal window that for example takes the value of the author field and when I press the button execute a function defined in the views.py that looks for the author and fill the field with the full name without closing the modal modal?

Thank you

Reply • Share ›

Kingsley Kanu 'SuGe' • 6 months ago

This is more than amazing....! Its ecstatic!!!!! Been looking for such. Just like in java. Thanks a lot.

Reply • Share ›

Elder Marco • 6 months ago

I'm learning Django. Thank you very much Vitor!

Reply • Share ›

Tosin • 6 months ago

Thank you one million times for this awesome tutorial. You are wonderful!
I have this little observation. The Ajax call was not rendering well on my browsers (Firefox and Chrome). It was just delivering raw dict like syntax on the page. I solved the issue by using event.preventDefault. Please, i will recommend that you include that to save other beginners like me the headache...
Thanks

Reply • Share ›

miraj naik • 7 months ago

thanks ...but i am trying to create form but its not appear

<label for="{{ field.id_for_label }}">{{ field.label }}</label>
its showing error

Reply • Share ›

willian firmino • 7 months ago

This is a great article, how about a second part using djangorestframework that would be very nice.

Reply • Share ›

अंकुर बेलबासे • 7 months ago

Amazing...thanks!!! man very helpful.!!!

Reply • Share ›

Saroj Rai • 7 months ago

Thank you....its very helpful !

Reply • Share ›

sohamnavadiya • 9 months ago

Thanks for sharing this tutorial. Very help full. Keep it up and share tutorial like this

Reply • Share ›

Vitor Freitas Mod → sohamnavadiya • 9 months ago

Thanks a lot!! :-)

Reply • Share ›

Diego González Espinilla • 9 months ago

Amazing tutorial .. Thank you very much Vitor.

How can i add a loading or waiting image or waiting message while executing a long process?

For example: http://www.ajaxload.info/

Thanks.

Regards.

Reply • Share ›

Vitor Freitas Mod → Diego González Espinilla • 9 months ago

Hi Diego,
It depends on where you want to show the loading gif / waiting image. In some cases I like to put the loading image in a hidden div, and then just display it when the processing starts, and hide it again when it completes.
Either way, normally this is what you need to do:

```
$.ajax({
  url: '/upload/',
  beforeSend: function () {
    $("#loading").show();
  },
  success: function (data) {
    // do something...
  }
  complete: function () {
    $("#loading").hide();
  }
});
```

^ | v • Reply • Share ›

Diego González Espinilla ➦ Vitor Freitas • 9 months ago

Thank you very much Vitor, works like a charm!!!

Other question:

In template:

```
{% for delegado in datos.delegados %}
  <tr>
    <td>
      <input type="checkbox" name="list_delegs" value="{{ delegado }}"/></td>
    </td>
    <td>{{ delegado }}</td>
  </tr>
{% endfor %}
```

How can i send all checked checkboxes to multiple delete as your previous code.

I don´t know how pass list of checked checkboxes instead of ok to url like:

```
url(r'^books/(?P<pk>\d+)/delete/$', views.book_delete, name='book_delete')
```

and in views.py how to get list of checked checkboxes

Thanks.