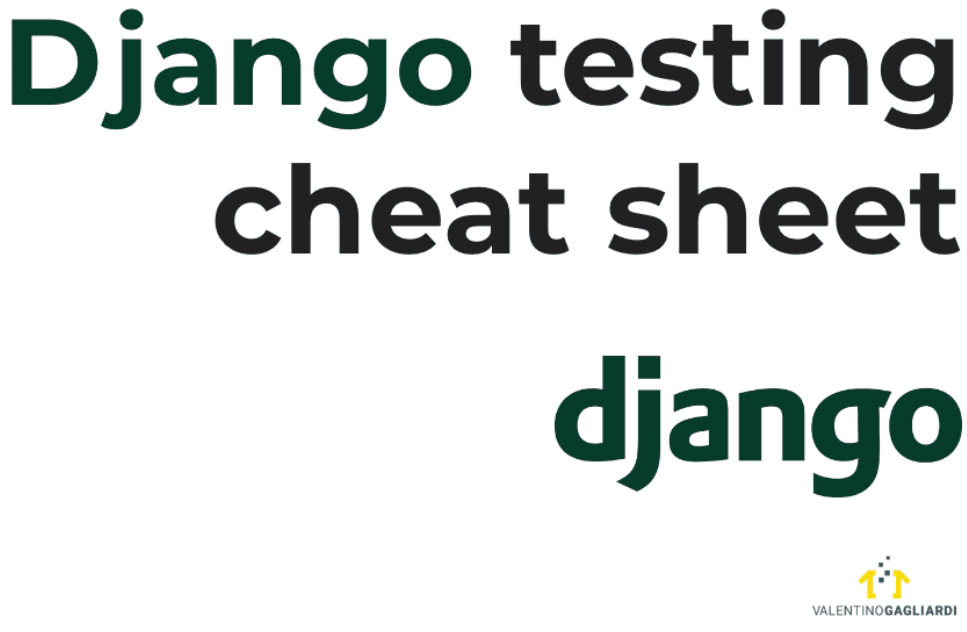
Last updated: May 4, 2020 by Valentino Gagliardi - 14 minutes read

Django Testing Cheat Sheet

A cheat-sheet of common testing patterns and best practices in Django applications.



DISCLAIMER: the examples presented here might not necessarily be "the right way" to do X in Django. Get in touch to propose changes or addictions.

TABLE OF CONTENTS

To follow along make sure to create a new Django project. With the project in place create a Django app named **library**:

```
django-admin startapp library
```

Next up enable the app in `settings.py`:

```
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    # enable the app
    "library.apps.LibraryConfig",
]
```

Prelude: How do I know what to test?

Use coverage. Install the package in your Django project:

```
pip install coverage
```

Run the tool inside the project folder:

```
coverage run --omit='*/venv/*' manage.py test
```

After the first pass you can get a coverage report with:

```
coverage report
```

You can also generate an HTML report with (a new folder called htmlcov will appear inside the project root):

```
coverage html
```

Test organization

Test organization is hard, and depends heavily on team preferences. A good starting point is **to split at least your test files**.

Instead of a single `tests.py` in the app folder you can create a `tests` folder where each file holds tests for a single facet of the application:

```
library/
├── admin.py
├── apps.py
├── __init__.py
├── migrations
│   └── __init__.py
├── models.py
├── tests
│   ├── api.py
│   ├── __init__.py
│   └── models.py
│   └── web.py
└── views.py
```

Here you can see a `tests` folder with `api.py` and `web.py`. `api.py` holds tests for API endpoints, while `web.py` can test regular HTML pages. You can also have a `models.py` for testing models.

Another common approach is to have a `feature` folder where each file has tests for a single app's feature:

```
library/
├── admin.py
├── apps.py
├── __init__.py
├── migrations
│   └── __init__.py
├── models.py
├── tests
│   ├── features
│   │   ├── search
│   │   ├── search_api
│   │   └── user_profile
│   ├── __init__.py
└── views.py
```

In **this guide we'll use the first approach**.

Testing a many to many relationship

Scenario: testing two related models.

Consider two models: **Book** and **Author**. A book can have many authors, and an author can have many books connected. To express this relationship we

can apply a `ManyToManyField` from **Book** to **Author**.

If you want to follow along create the models in `library/models.py`:

```
from django.db import models


class Author(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)


class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(to=Author)
```

Then run and apply the migration:

```
python manage.py makemigrations library
python manage.py migrate
```

As a first test we might want to check that we didn't forget to add `ManyToManyField`. Given a Book we want to count one or more Authors.

In a file named `library/tests/models.py` we can create the following test:

```
from django.test import TestCase
from library.models import Author, Book
```

```
class TestModels(TestCase):
    def test_book_has_an_author(self):
        book = Book.objects.create(title="The man in the high
        philip = Author.objects.create(first_name="Philip", la
        juliana = Author.objects.create(first_name="Juliana",
        book.authors.set([philip.pk, juliana.pk])
        self.assertEqual(book.authors.count(), 2)
```

Here we create one book and two authors. To assign our authors to the book we do:

```
book.authors.set([philip.pk, juliana.pk])
```

We could also do the opposite, assign the book to each author:

```
from django.test import TestCase
from library.models import Author, Book


class TestModels(TestCase):
    def test_book_has_an_author(self):
        book = Book.objects.create(title="The man in the high
        philip = Author.objects.create(first_name="Philip", la
        juliana = Author.objects.create(first_name="Juliana",
        philip.book_set.add(book)
        juliana.book_set.add(book)
        self.assertEqual(book.authors.count(), 2)
```

To run the test import `library/tests/models.py` in `library/tests/__init__.py`:

```
from .models import *
```

Then run:

```
python manage.py test library
```

Resources:

- Many-to-many relationships
- Django TestCase

Testing model str

Scenario: testing a model string representation.

Django models may have a `__str__` method which drives how the model is represented as a string. Consider again our models:

```
from django.db import models


class Author(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
```

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(to=Author)
```

To display Author as first_name + last_name and Book with its title we can add the corresponding `__str__` method to each model:

```
from django.db import models


class Author(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)

    def __str__(self):
        return f"{self.first_name} {self.last_name}"


class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(to=Author)

    def __str__(self):
        return self.title
```

In the file named `library/tests/models.py` we can add the following test:

```
from django.test import TestCase
from library.models import Author, Book


class TestModels(TestCase):
    def test_model_str(self):
        book = Book.objects.create(title="The man in the high
        philip = Author.objects.create(first_name="Philip", la
        self.assertEqual(str(book), "The man in the high castl
        self.assertEqual(str(philip), "Philip K. Dick")

        # More tests here
```

To run the test import `library/tests/models.py` in `library/tests/__init__.py`:

```
from .models import *
```

Then run:

```
python manage.py test library
```

Testing model fields en masse

Scenario: testing "crowded" models.

Consider a crowded Django model with many fields:

```
class Event(models.Model):
    title = models.CharField(max_length=60)
    seo_title = models.CharField(max_length=59)
    seo_description = models.CharField(max_length=160)
    abstract = models.CharField(max_length=160)
    body = models.TextField(default="")
    duration = models.IntegerField(default=0)
    slug = models.SlugField(max_length=20)
    start_date = models.DateTimeField()
    end_date = models.DateTimeField()
    price = models.IntegerField()
    location = models.TextField(max_length=100)
    created_at = models.DateTimeField(auto_now_add=True)
    published = models.BooleanField(default=False)

    def __str__(self):
        return f"{self.title}"
```

Would be unpractical to populate each field by hand in a test:

```
from django.test import TestCase
from library.models import Author, Book, Event
from datetime import datetime


class TestModels(TestCase):
    def test_event_model(self):
        event = Event.objects.create(
```

```
            title="Some title",
            seo_title="Some Seo title",
            seo_description="Some description",
            abstract="The abstract",
            body="The body",
            duration=2,
            slug="the-slug",
            start_date=datetime.now(),
            end_date=datetime.now(),
            price=800,
            location="Rome",
            published=False,
        )
```

Instead with a tool like Model bakery you can delegate field creation.
Install Model bakery with:

```
pip install model_bakery
```

Then in your test:

```
from django.test import TestCase
from library.models import Author, Book, Event
from model_bakery import baker


class TestModels(TestCase):
    def test_event_model(self):
        event = baker.make(Event, title="The man in the high c
        self.assertEqual(str(event), "The man in the high cast
```

```
        # More tests here
```

You can pass your own fields if you need to override them. Model bakery is also convenient for generating huge amounts of models. See Jeff's post below.

Resources:

- Model bakery
- How do I generate 1,000 objects in Django and DRF to test?

Testing a POST request

Scenario: accept POST requests on "/contacts/" with an HTML form.

Let's say you want to create a **contact form for your library with Django** to get contacts from students. First thing first you may want to write a test for it.

Following the test structure we made, create a new file in `library/tests/web.py`. In this file we can import `TestCase`, our model, and write a skeleton for the test:

```
from django.test import TestCase
from .models import Contact


class TestStudentContactForm(TestCase):
```

```
    def test_can_send_message(self):
        pass
```

Now to test this feature we need to create some **data to send alongside with the POST request**. Note that the data should match the model's fields.

So given an hypothetical model like this:

```
class Contact(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    message = models.TextField(max_length=400)

    def __str__(self):
        return f"{self.first_name} {self.last_name}"
```

in the test we can write a data dictionary:

```
from django.test import TestCase
from library.models import Contact


class TestStudentContactForm(TestCase):
    def test_can_send_message(self):
        data = {
            "first_name": "Juliana",
            "last_name": " Crain",
            "message": "Would love to talk about Philip K. Dic
        }
```

Now with **Django test client we send the request**. As a first test we can **check if a Contact instance is created in the database**:

```
from django.test import TestCase
from library.models import Contact


class TestStudentContactForm(TestCase):
    def test_can_send_message(self):
        data = {
            "first_name": "Juliana",
            "last_name": " Crain",
            "message": "Would love to talk about Philip K. Dic
        }
        response = self.client.post("/contact/", data=data)
        self.assertEqual(Contact.objects.count(), 1)
```

Next up we can check if the view **redirected correctly**:

```
from django.test import TestCase
from library.models import Contact


class TestStudentContactForm(TestCase):
    def test_can_send_message(self):
        data = {
            "first_name": "Juliana",
            "last_name": " Crain",
            "message": "Would love to talk about Philip K. Dic
```

```
        }
        response = self.client.post("/contact/", data=data)
        self.assertEqual(Contact.objects.count(), 1)
        self.assertRedirects(response, "/thanks/")
```

This is a test for the classic **POST/Redirect/GET pattern** so common in web development.

Pay attention because the test above **skips entirely the HTML form**. You might also want to test the template (or at least a couple of HTML input):

```
from django.test import TestCase
from library.models import Contact


class TestStudentContactForm(TestCase):
    def test_can_send_message(self):
        data = {
            "first_name": "Juliana",
            "last_name": " Crain",
            "message": "Would love to talk about Philip K. Dic
        }
        response = self.client.get("/contact/")
        self.assertTemplateUsed(response, "library/contact_for
        self.assertContains(response,"first_name")
        self.assertContains(response, "last_name")
        response = self.client.post("/contact/", data=data)
        self.assertEqual(Contact.objects.count(), 1)
        self.assertRedirects(response, "/thanks/")
```

Here we test for appearance two model fields in the HTML:

```
        self.assertContains(response,"first_name")
        self.assertContains(response, "last_name")
```

Testing a couple of fields is enough as long as you include the form in the HTML. Another thing to keep in mind is that the test client **skips CSRF validation**. Don't **forget to include the token in a template**! For a more realistic test you can also use Selenium or Splinter.

To run the test import `library/tests/web.py` in `library/tests/__init__.py`:

```
from .web import *
```

Then run:

```
python manage.py test library
```

To make this test pass you can use a Django CreateView as described here.

Resources

- POST/Redirect/GET pattern
- Django Test client

More robust URLs in tests

It's ok-ish to call URLs in tests this way:

```
response = self.client.post("/contact/", data=data)
```

Better, you can use `reverse` to avoid brittle tests:

```
from django.test import TestCase
from library.models import Contact
from django.urls import reverse


class TestStudentContactForm(TestCase):
    def test_can_send_message(self):
        data = {
            "first_name": "Juliana",
            "last_name": " Crain",
            "message": "Would love to talk about Philip K. Dic
        }
        response = self.client.get(reverse("contact"))
        self.assertTemplateUsed(response, "library/contact_for
        self.assertContains(response, "first_name")
        self.assertContains(response, "last_name")
        response = self.client.post(reverse("contact"), data=d
        self.assertEqual(Contact.objects.count(), 1)
        self.assertRedirects(response, reverse("thanks"))
```

Now you can reference **URLs by name rather than by path**, as long as you name your URLs in the corresponding `urls.py`. For example an hypothetical `library/urls.py` would look like this:

```
from django.urls import path
from .views import ContactCreate, thanks

urlpatterns = [
    path("contact/", ContactCreate.as_view(), name="contact"),
    path("thanks/", thanks, name="thanks"),
]
```

Check this post for a complete example.

Providing data dictionary from a Django model

There are situations where you want to test a model instance in the same block with a POST request. To avoid duplication you can use `model_to_dict` on the model instance:

```
from django.test import TestCase
from library.models import Contact
from django.urls import reverse
from django.forms.models import model_to_dict


class TestStudentContactForm(TestCase):
    def test_can_send_message(self):
        contact = Contact.objects.create(
            first_name="Juliana",
            last_name="Crain",
```

```
            message="Would love to talk about Philip K. Dick",
        )
        self.assertEqual(str(contact), "Juliana Crain")
        ## Convert the model to a dictionary
        data = model_to_dict(contact)
        # Post
        response = self.client.post(reverse("contact"), data=d
        self.assertRedirects(response, reverse("thanks"))
```

Thanks to Augusto for this tip.

Testing authentication

Scenario: show the "/download/" page only to authenticated users.

We have a URL "/download/" connected to a view. **Only authenticated users should access this view**. As a first test we can check that any anonymous user is redirected to the login page defined in `settings.LOGIN_URL` (defaults to `/accounts/login/` followed by `?next=/download/`):

```
class TestDownloadView(TestCase):
    def test_anonymous_cannot_see_page(self):
        response = self.client.get(reverse("download"))
        self.assertRedirects(response, "/accounts/login/?next=
```

Authenticated users instead can access the page. To test an authenticated user we create the user in the test block, and with `client.force_login()` we let it pass:

```
from django.test import TestCase
from django.urls import reverse
from django.contrib.auth.models import User


class TestDownloadView(TestCase):
    def test_anonymous_cannot_see_page(self):
        response = self.client.get(reverse("download"))
        self.assertRedirects(response, "/accounts/login/?next=

    def test_authenticated_user_can_see_page(self):
        user = User.objects.create_user("Juliana," "juliana@de
        self.client.force_login(user=user)
        response = self.client.get(reverse("download"))
        self.assertEqual(response.status_code, 200)
        # Or assert you can see stuff on the page
```

Note that you should swap `from django.contrib.auth.models import User` with any custom Django user, if present.

Resources:

- Limiting access to logged-in users

Testing request headers

Scenario: we want to test how Django behaves depending on a request header.

This scenario is useful for **testing a Django middleware**, or any view that takes decisions **depending on a request header**. Consider the following view:

```
def index(request):
    if not request.META["HTTP_HOST"] == "www.my-domain.dev":
        return HttpResponse("Wrong host!")
    return HttpResponse("Correct host!")
```

It returns two different responses depending on `HTTP_HOST`'s value. A more robust version with `get_host`:

```
def index(request):
    if not request.get_host() == "www.my-domain.dev":
        return HttpResponse("Wrong host!")
    return HttpResponse("Correct host!")
```

A first test for the view can check if the response contains "Wrong host!" when `HTTP_HOST` is not specified:

```
from django.test import TestCase
from django.urls import reverse


class TestHostHeader(TestCase):
    def test_empty_host(self):
        response = self.client.get(reverse("index"))
        self.assertContains(response, "Wrong host!")
```

With another test we can check for "Wrong host!" if the `HTTP_HOST` is not "www.my-domain.dev". There are two ways for passing `HTTP_HOST`. Option one:

```
    def test_wrong_host(self):
        response = self.client.get(reverse("index"), HTTP_HOST
        self.assertContains(response, "Wrong host!")
```

Here `client.get()` accepts extra keyword arguments. Option two:

```
    def test_wrong_host_construct(self):
        client = Client(HTTP_HOST="www.wrong-domain.dev")
        response = client.get(reverse("index"))
        self.assertContains(response, "Wrong host!")
```

Here we construct the client with a custom header. **Both are valid options**. For a bit of context here are the three tests:

```
from django.test import TestCase, Client
from django.urls import reverse


class TestHostHeader(TestCase):
    def test_empty_host(self):
        response = self.client.get(reverse("index"))
        self.assertContains(response, "Wrong host!")

    def test_wrong_host(self):
```

```
        response = self.client.get(reverse("index"), HTTP_HOST
        self.assertContains(response, "Wrong host!")

    def test_wrong_host_construct(self):
        client = Client(HTTP_HOST="www.wrong-domain.dev")
        response = client.get(reverse("index"))
        self.assertContains(response, "Wrong host!")
```

Finally, we can test for "Correct host!" by passing the expected `HTTP_HOST` in another test (again, pick your own style for passing the header):

```
    def test_correct_host(self):
        response = self.client.get(reverse("index"), HTTP_HOST
        self.assertContains(response, "Correct host!")
```

The complete test suite:

```
from django.test import TestCase, Client
from django.urls import reverse


class TestHostHeader(TestCase):
    def test_empty_host(self):
        response = self.client.get(reverse("index"))
        self.assertContains(response, "Wrong host!")

    def test_wrong_host(self):
        response = self.client.get(reverse("index"), HTTP_HOST
```

```
        self.assertContains(response, "Wrong host!")

    def test_wrong_host_construct(self):
        client = Client(HTTP_HOST="www.wrong-domain.dev")
        response = client.get(reverse("index"))
        self.assertContains(response, "Wrong host!")

    def test_correct_host(self):
        response = self.client.get(reverse("index"), HTTP_HOST
        self.assertContains(response, "Correct host!")
```

A common use case for this test is a single Django project serving requests for multiple domain names, where each domain must load one and only Django app.

Resources:

- HttpRequest META
- How to handle multiple sites (virtual hosts) in Django

Django REST framework interlude

Django REST framework (**DRF from now on**) is a fantastic Django tool for building RESTful APIs. To install Django REST framework in your project run:

```
pip install djangorestframework
```

Next up enable DRF in `settings.py`:

```
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "library.apps.LibraryConfig",
    # Enable Django REST
    "rest_framework",
]
```

DRF offers a group of custom testing classes over Django's `TestCase` or `LiveServerTestCase`. `APITestCase` is the go-to class for testing DRF endpoints.

DRF: Testing POST requests

Scenario: accept POST requests on a API endpoint at "api/contacts/".

To test your API you can create a new file in `library/tests/api.py` with a skeleton for the test:

```
from rest_framework.test import APITestCase
from library.models import Contact
from django.urls import reverse


class TestContactAPI(APITestCase):
```

```
    def test_post_request_can_create_new_entity(self):
        pass
```

To test this feature we need to create some **data to send alongside with the POST request**. Note that the data should match the model's fields.

To **make this test pass in DRF you need**:

- a model
- a model serializer
- a `CreateAPIView` and the corresponding URL

Instruction for working with Django REST Framework are outlined here.

So given a hypothetical model like:

```
class Contact(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    message = models.TextField(max_length=400)

    def __str__(self):
        return f"{self.first_name} {self.last_name}"
```

We can test like so:

```
from rest_framework.test import APITestCase
from library.models import Contact
from django.urls import reverse
```

```
class TestContactAPI(APITestCase):
    def test_post_request_can_create_new_entity(self):
        data = {
            "first_name": "Juliana",
            "last_name": " Crain",
            "message": "Would love to talk about Philip K. Dic
        }
        self.client.post(reverse("contact_create"), data=data)
        self.assertEqual(Contact.objects.count(), 1)
```

There's not so much to test in a simple case like this, but a check for 201 won't harm if you're paranoid like me:

```
from rest_framework.test import APITestCase
from rest_framework import status
from library.models import Contact
from django.urls import reverse


class TestContactAPI(APITestCase):
    def test_post_request_can_create_new_entity(self):
        data = {
            "first_name": "Juliana",
            "last_name": " Crain",
            "message": "Would love to talk about Philip K. Dic
        }
        response = self.client.post(reverse("contact_create"),
```

```
        self.assertEqual(response.status_code, status.HTTP_201
        self.assertEqual(Contact.objects.count(), 1)
```

To run the test import `library/tests/api.py` (and any previous test you wrote) in `library/tests/__init__.py`:

```
from .models import *
from .web import *
from .api import *
```

Then to run only the API test:

```
python manage.py test library.tests.api
```

DRF: Testing authentication

Scenario: accept GET requests on a API endpoint at "api/secret/" only for authenticated users.

We have an endpoint "api/secret/" connected to a DRF ListView. **Only authenticated users should access this view**. As a first test we can check that any anonymous user gets a 403 forbidden:

```
from rest_framework.test import APITestCase
from rest_framework import status
from django.urls import reverse
```

```
class TestContactAPI(APITestCase):
    def test_anonymous_cannot_see_contacts(self):
        response = self.client.get(reverse("contact_view"))
        self.assertEqual(response.status_code, status.HTTP_403
```

A minimal view to make the test pass can be:

```
from rest_framework.generics import ListAPIView
from library.serializers import ContactSerializer
from library.models import Contact
from rest_framework.authentication import SessionAuthenticatio
from rest_framework.permissions import IsAuthenticated


class ContactViewAPI(ListAPIView):
    authentication_classes = [SessionAuthentication]
    permission_classes = [IsAuthenticated]
    serializer_class = ContactSerializer
    queryset = Contact.objects.all()
```

This view assumes session authentication with the API being called in the same context of an hypothetical JavaScript frontend. In a decoupled architecture you would use token based authentication.

Authenticated users instead can access the page. To test an authenticated user we create the user in the test block, and with `client.force_login()` we let it pass:

```
from rest_framework.test import APITestCase
from rest_framework import status
from django.urls import reverse
from django.contrib.auth.models import User


class TestContactAPI(APITestCase):
    def test_anonymous_cannot_see_contacts(self):
        response = self.client.get(reverse("contact_view"))
        self.assertEqual(response.status_code, status.HTTP_403

    def test_authenticated_user_can_see_contacts(self):
        user = User.objects.create_user("Juliana," "juliana@de
        self.client.force_login(user=user)
        response = self.client.get(reverse("contact_view"))
        self.assertEqual(response.status_code, status.HTTP_200
        # Or assert the JSON response
```

Note that you should swap `from django.contrib.auth.models import User` with any custom Django user, if present.

Resources:

- Setting the authentication scheme on DRF

MORE

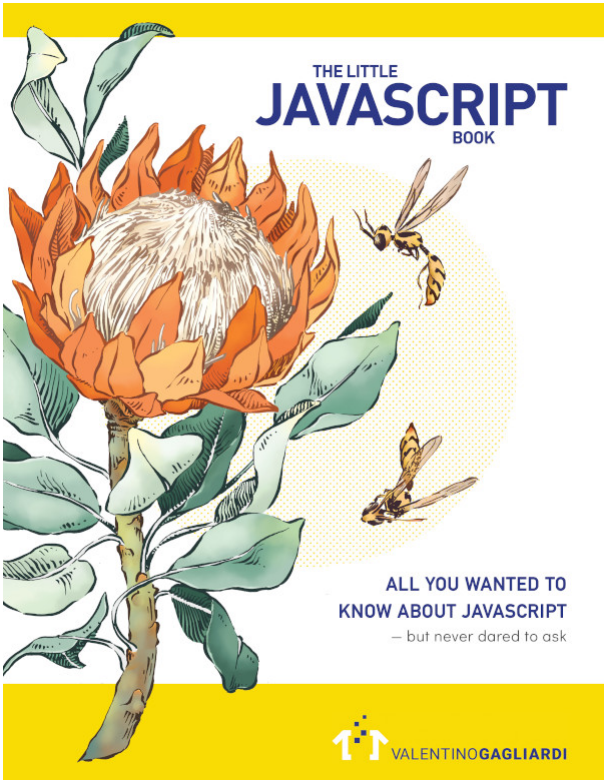
More is coming soon. Stay tuned!

STAY UPDATED

Be the first to know when I publish new stuff.

BOOKS

The Little JavaScript Book



Hi! I'm Valentino! Educator and consultant, I help people learning to code with on-site and remote workshops. Looking for JavaScript and Python training? Let's get in touch!

More from the blog:

- Django Tips: Recovering Gracefully From ORM Errors
- Understanding many to one in Django
- Tutorial: Django REST with React (Django 3 and a sprinkle of testing)
- Working with request.data in Django REST framework
- How to create a contact form with Django, widget customization
- How to handle multiple sites (virtual hosts) in Django
- How to create a Django project and a Django application
- Asynchronous tasks in Django with Django Q
- How to create a Django project from a template
- Building a Django middleware (injecting data into a view's context)