

Django Unstuck

Suggestions for common challenges in your projects



About me

I'm Johannes Spielmann, software developer-for-hire from Germany.
I've been doing Django projects for almost 15 years.
I've done projects both small and large, both in commercial and non-commercial settings
I've seen all of what we're about to do many times.
You should hire me!

Email me at j@spielmannsolutions.com!

Django Unstuck: Suggestions for common challenges in your projects

There are challenges that
come up in every Django
project.


How do we get past them?

Here are some solutions I've found

Some people may call these things “best practices” or “patterns” or “recipes” or “ideas” or “opinions”.

These are just things I've found to work in these situations many times.

Some of these solutions are more directly applicable, others are more guideline-style.



App management Templates Middlewares and Context Processors Code in models, views or managers?

We are going to look at these four things:

App management and placement (and urlpatterns and settings)

How do we make sure we can find and understand our apps? How do we make it easy to decide where code goes?

How do we make our settings modular and easy to understand and modify?

Templates

Where do we put them? What do we call our blocks?

When to use Middlewares and context processors and what are they?

Code/Logic can go into many different places: models, views, managers or somewhere else entirely? So where should you put your logic then?

There are more...

App management and placement (and urlpatterns)

Username vs email address

Registration

Background tasks and long-running processes and Caching

Templates: Placement, folders, blocks and inheritance and namespaces

Should you do i18n and l10n right away?

When and how to start caching (memcached, redis etc.)

Fat models, fat views or fat managers or something else?

When to use Middlewares and context processors and what are they?

How to secure access: security middlewares or login_required (white vs black list)

How to create files and store them in file models

What to do about image scaling and thumbnailing (and hosting)?

How to serve content: coded pages, flatpages or Wagtail?

Join us!

<https://github.com/shezi/django-unstuck>

<https://discord.gg/bUsu9B6Ek6>

<https://t.me/djangoRhein>

<https://github.com/shezi/django-unstuck>

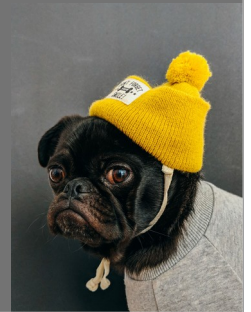
<https://discord.gg/bUsu9B6Ek6>

<https://t.me/djangoRhein>

Part 1

App management and
urlpatterns and settings





Let's start a new Django project



```
$ django-admin.py startproject tuque  
$ cd tuque  
$ ./manage.py startapp mytuque
```

Note: I took that name from a name generator. It's a woolen hat.
Image credit: <https://unsplash.com/photos/AQRp2NH-O8k>

Challenge: organize your project such that:

-  everything has its place
-  what belongs together, stays together
-  code/URLs are easily found
-  settings are flexible

Step 1 configuration



Create a config directory

Below that, a settings directory

Put the configuration in config directory

Put your config in config/settings/

Create base config, development config, production config (and optionally local config, but do not forget to .gitignore that one!)

Wire it all up so that it works.

Step 2

apps directory



Create an apps directory
Put the python path into manage.py and wsgi.py
Rejoice in the cleanness of your construction

Step 3

URL patterns



Put an `urls.py` into your new app, give it an `app_name` and `urlpatterns`.
Wire it up directly in `config/urls.py`



Step 4

template placement

Actually, no, that's in Part 2

Part 2

Templates: Placement,
folders, blocks and
inheritance and namespaces

Where are your templates?
What are they called?

Challenge

All templates live in one big namespace.
Yet there are many places in the filesystem.

Decide where a template file goes!
Identify what a template belongs to!
Make sure namespace clashes are minimal!

Step 1

Main template directory

Contains base.html and sub-bases, and global utilities (menu snippets, breadcrumbs, form snippets, ...) -- everything that is part of the project as a whole.

Do not forget to wire it up in settings!

Note: easy to include, always at the same place (in template space and file system)

Step 2

In-app template directories

These folders contain everything for that specific app. Each item in here should be tied either directly to a view or to some other template.

Always inside a sub-folder with the app name, never "on top".

Always named like the view function/class, and the URI pattern should have the same name. That way you can always find one from the other.

You should never need to include things from another app. If that is the case, think about whether you should migrate that specific thing to the main directory.

Now do the same for static files!

Static files are very similar to template files: have one global folder “static/” in your project root, where project-wide things go: CSS, JS, downloadable files etc. Then each app gets its own static folder, with an app-name folder inside. This reduces clashes and makes files easy to find and understand.

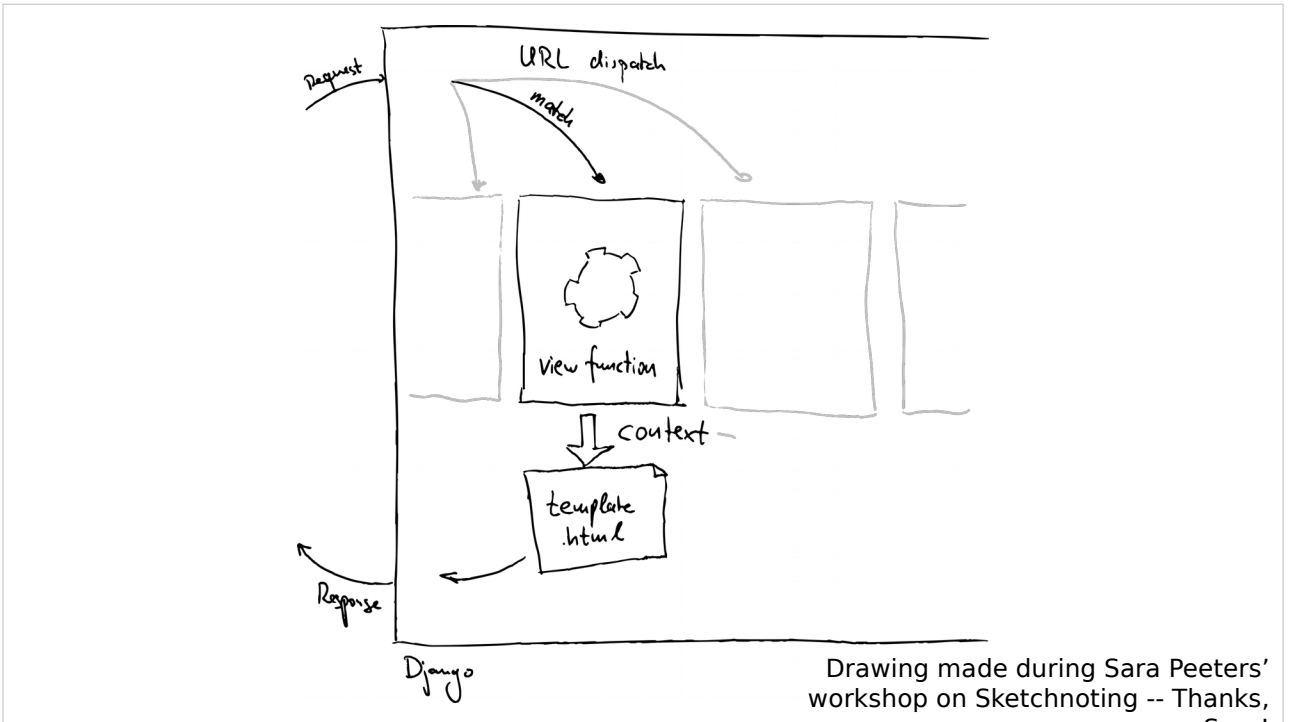
Part 3

Middlewares and context processors (and custom template tags)

What are they?

When do we use each?

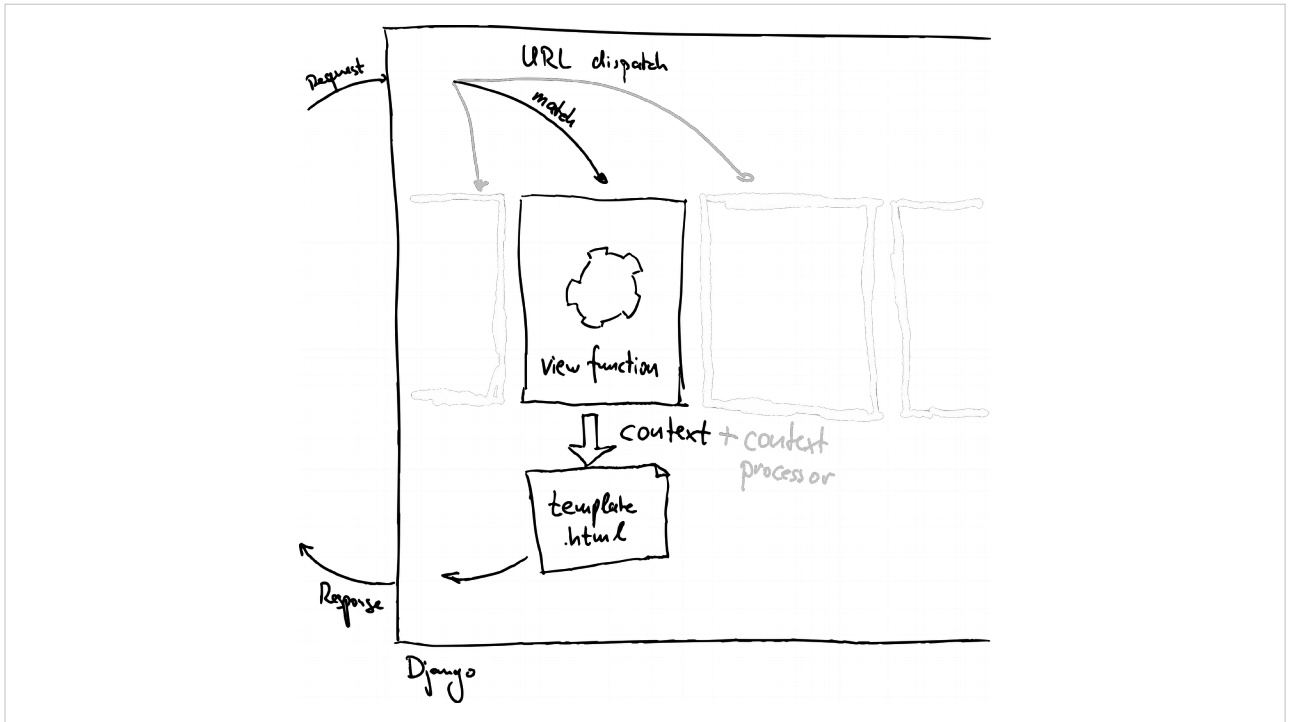
Note: This is not an implementation guide, those exist aplenty. This is a decision guide: which one should you implement?



The request cycle: a request comes in, gets processed by a view, rendered in a template
The result is a response.
Easy!

Context processors

Sometimes you want to do things that show up in each rendered template. That's what context processors do.



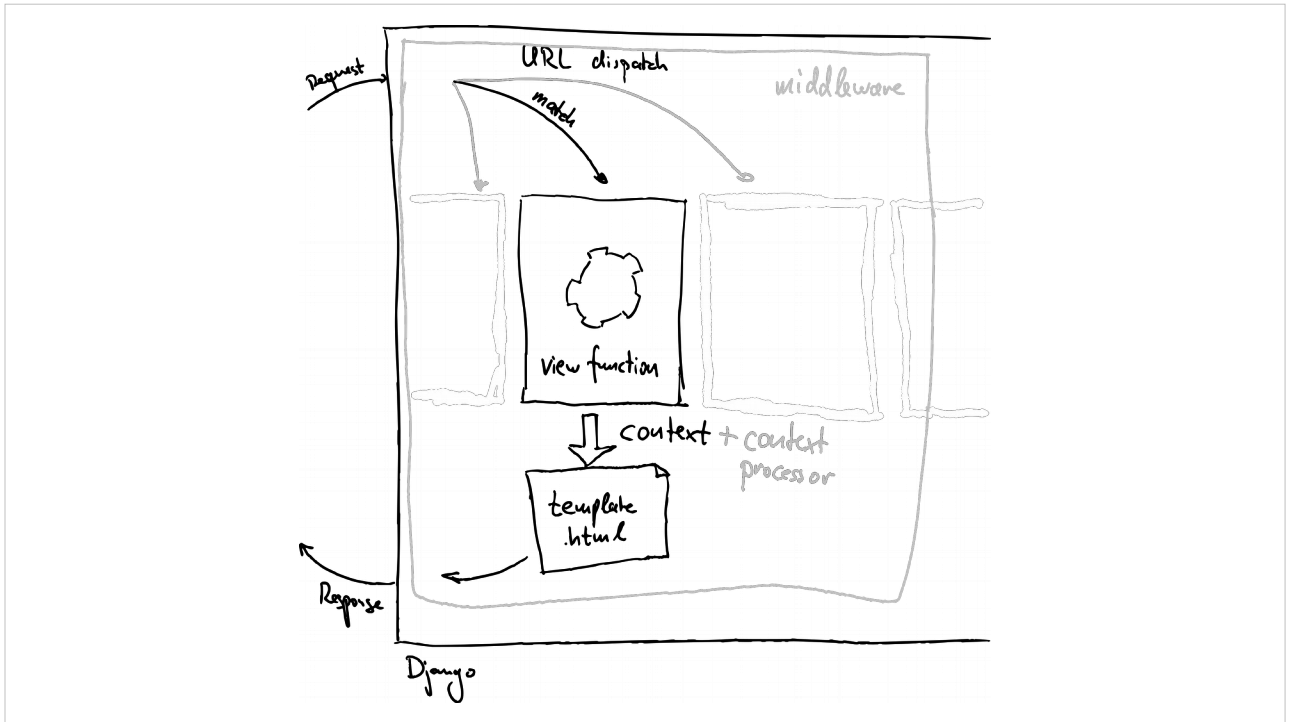
The request cycle: a request comes in, gets processed by a view, rendered in a template
The result is a response.
Easy!

```
def menu_processor(request):  
    return {  
        "menu_categories": ProductCategories.objects.all()  
    }
```

A simple code example: Fetch product categories from the database, so you can render a menu on each page.

Middleware

Finally, a middleware wraps the entire request/response cycle, so it can change every step of the process. This makes it more powerful, but also more complicated.



The request cycle: a request comes in, gets processed by a view, rendered in a template
The result is a response.
Easy!

```

class MessageMiddleware(MiddlewareMixin):
    """
    Middleware that handles temporary messages.
    """

    def process_request(self, request):
        request._messages = default_storage(request)

    def process_response(self, request, response):
        """
        Update the storage backend (i.e., save the messages).
        Raise ValueError if not all messages could be stored and DEBUG is True.
        """
        # A higher middleware layer may return a request which does not contain
        # messages storage, so make no assumption that it will be there.
        if hasattr(request, '_messages'):
            unstored_messages = request._messages.update(response)
            if unstored_messages and settings.DEBUG:
                raise ValueError('Not all temporary messages could be stored.')
        return response

```

An example from the Django code base: the message middleware from the messages framework.

The code itself is not important. What's important here is the structure: a middleware is a class that has hooks into the request/response lifecycle.

Here, two hooks are used: `process_request` and `process_response`. But there are more. For details, see

<https://docs.djangoproject.com/en/3.2/topics/http/middleware/#other-middleware-hooks>

When should you use which
one?

Menu structure

Context processor

Content-Security-Policy headers

Middleware

<https://django-csp.readthedocs.io/>

Detecting problematic DB queries

Middleware

<https://github.com/jmcarp/nplusone>

Counting db queries

Middleware

<https://github.com/bradmontgomery/django-querycount>

Show shopping basket

Context processor

Monitoring login attempts

Middleware

<https://github.com/jazzband/django-axes>

CORS headers

Middleware

<https://github.com/adamchainz/django-cors-headers>

Show most recent blog posts

Context processor

Monitoring server errors

Middleware

<https://raven.readthedocs.io/en/stable/integrations/django.html>

Highlight items on sale

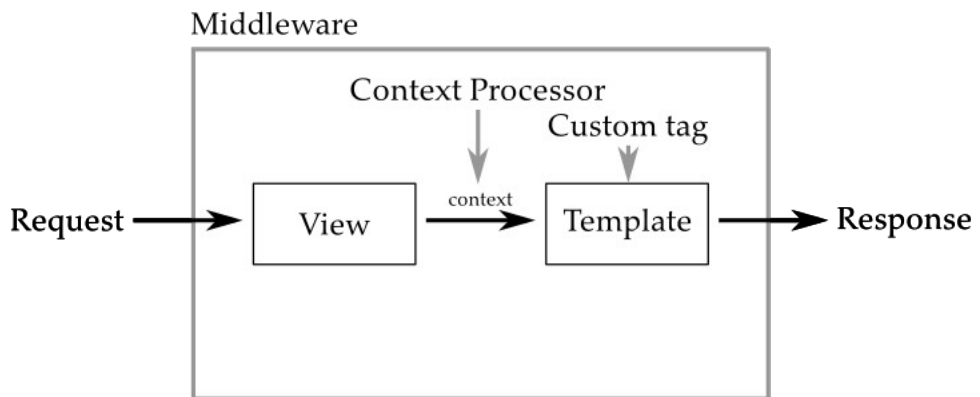
Context processor

Breadcrumbs

???

Show some items in dynamic
page footer

Context processor



This is the main takeaway of this section:
If you do something for every rendered *template*, use a context processor.
If you do something for every *request*, use a middleware.

Part 4

Code in models, views,
managers or somewhere
else?

Challenge: Where do you put your code?

We have some models, and these models need to do some things.

Where should we put the code for that?

Into the models

Into a model controller

Into the view

Into some other arbitrary place

Note: This is ALSO not an implementation tutorial, but a decision tutorial.

Note: there is no real answer!

Models

Put everything in your model class.

- model classes are supposed to be storage abstractions
- no single purpose for model class
- + everything that has to do with data is in one place

We call these “massive models”.

```

class Subscriber(models.Model):
    user_rec = models.ForeignKey(User)
    address_one = models.CharField(max_length=100)
    address_two = models.CharField(max_length=100, blank=True)
    city = models.CharField(max_length=50)
    state = models.CharField(max_length=2)
    stripe_id = models.CharField(max_length=30, blank=True)

    class Meta:
        verbose_name_plural = 'subscribers'

    def __str__(self):
        return u"%s's Subscription Info" % self.user_rec

    def charge(self, request, email, fee):
        # Set your secret key: remember to change this to your live secret key
        # in production. See your keys here https://manage.stripe.com/account
        stripe.api_key = settings.STRIPE_SECRET_KEY

        # Get the credit card details submitted by the form
        token = request.POST['stripeToken']

        # Create a Customer
        stripe_customer = stripe.Customer.create(
            card=token,
            description=email
        )

        # Save the Stripe ID to the customer's profile
        self.stripe_id = stripe_customer.id
        self.save()

        # Charge the Customer instead of the card
        stripe.Charge.create(
            amount=fee, # in cents
            customer=self.stripe_id
        )

```

An example: a subscriber that has methods for putting charges on their card.

Not really a storage abstraction any more. There is no clear purpose for this class, as it interacts with the database as well as with other APIs.

Also, this class will end up having 2000 lines and 100 methods. It is massive!

Model Managers

Put some methods into model manager.

Model managers are abstractions for database access. Every model class has a default manager, which is `ModelClass.objects`

- a bit more inconvenient
- instances must be passed
- not very SOLID either: purpose is supposed to be storage retrieval
- + probably correct placement for creation/copy methods

```
class ChargeManager(models.Manager):
    ...

    def charge(self, instance, request, email, fee):
        # ...
        return

class Subscriber(models.Model):
    user_rec = models.ForeignKey(User)
    address_one = models.CharField(max_length=100)
    address_two = models.CharField(max_length=100, blank=True)
    city = models.CharField(max_length=50)
    state = models.CharField(max_length=2)
    stripe_id = models.CharField(max_length=30, blank=True)

    # create manager here
    charges = ChargeManager()

    class Meta:
        verbose_name_plural = 'subscribers'

    def __str__(self):
        return u"%s's Subscription Info" % self.user_rec
```

Separating out the previous example into a model manager is much cleaner.

Views

Let the view function do all the stuff.

Well, not all the stuff obviously: have auxiliary functions and classes and methods, of course!

- harder to reuse
- structure is ad-hoc, not easy to read
- + business logic is all in one place.

```
def charge(request, product_id):
    product = Product.objects.get(id=product_id)
    subscriber = request.user.subscriber
    email = request.user.email
    fee = product.fee

    # ...

    return render(request, "subscriptions/charge.html", {
        # ...
    })
```

Let the view function do all the stuff.

Well, not all the stuff obviously: have auxiliary functions and classes and methods, of course!

- harder to reuse
- structure is ad-hoc, not easy to read
- + business logic is all in one place.

Utility files/classes?

They exist, too, yes. I have them in each and every one of my projects, because it is just so hard to decide where code goes.

And sometimes that's the right place, too. For code that is needed in many different places or does not have a clear affiliation.

Examples

Some examples:

- Prepare some query: model or model manager, depends
- Create new model instance and save it: manager
- Create new model instance without saving: manager or model
- Create a new model instance from something else: manager or model
- Compute a value for a single model instance: model or view
- Compute a value for a single model instance depending on some value in the request: view or model method with request

Prepare a query

Manager

Some examples:

- Prepare some query: model or model manager, depends
- Create new model instance and save it: manager
- Create new model instance without saving: manager or model
- Create a new model instance from something else: manager or model
- Compute a value for a single model instance: model or view
- Compute a value for a single model instance depending on some value in the request: view or model method with request

Create new model instance And save it

Manager

Some examples:

- Prepare some query: model or model manager, depends
- Create new model instance and save it: manager
- Create new model instance without saving: manager or model
- Create a new model instance from something else: manager or model
- Compute a value for a single model instance: model or view
- Compute a value for a single model instance depending on some value in the request: view or model method with request

Create new model instance Without saving

Model

Some examples:

- Prepare some query: model or model manager, depends
- Create new model instance and save it: manager
- Create new model instance without saving: manager or model
- Create a new model instance from something else: manager or model
- Compute a value for a single model instance: model or view
- Compute a value for a single model instance depending on some value in the request: view or model method with request

Create new model instance From something else

Manager

Some examples:

- Prepare some query: model or model manager, depends
- Create new model instance and save it: manager
- Create new model instance without saving: manager or model
- Create a new model instance from something else: manager or model
- Compute a value for a single model instance: model or view
- Compute a value for a single model instance depending on some value in the request: view or model method with request

Compute a value For a single model instance

Model method

Some examples:

- Prepare some query: model or model manager, depends
- Create new model instance and save it: manager
- Create new model instance without saving: manager or model
- Create a new model instance from something else: manager or model
- Compute a value for a single model instance: model or view
- Compute a value for a single model instance depending on some value in the request: view or model method with request

Compute a value for a single model instance depending on some value in the request

View

Some examples:

- Prepare some query: model or model manager, depends
- Create new model instance and save it: manager
- Create new model instance without saving: manager or model
- Create a new model instance from something else: manager or model
- Compute a value for a single model instance: model or view
- Compute a value for a single model instance depending on some value in the request: view or model method with request

Decisions, decisions!

There is no right way to do it, but there's also no wrong way to do it.

Don't be afraid to try things out. Don't be afraid to go back and change or rearrange things.

Do write tests, they will give you confidence when rearranging your code!

That's all!

There's more!

<https://github.com/shezi/django-unstuck>

<https://discord.gg/bUsu9B6Ek6>

<https://t.me/djangoRhein>

<https://github.com/shezi/django-unstuck>

<https://discord.gg/bUsu9B6Ek6>

<https://t.me/djangoRhein>

My name is Johannes Spielmann. Email me at j@spielmannsolutions.com!
I'm looking forward to talking to all of you!