
Django Tips

ChillarAnand

Aug 19, 2019

CONTENTS

1	Preface	1
1.1	Why this book?	1
1.2	Who should read this book?	1
1.3	Acknowledgements	1
2	Auto Register All Models In Admin	3
2.1	Manual Registration	3
2.2	Auto Registration	4
2.3	Auto Registration With Fields	6
3	Custom Admin Actions For Querysets & Individual Objects	9
3.1	Custom Actions On Querysets	9
3.2	Custom Actions On Individual Objects	10
4	Hyperlink Foreignkeys To Its Change View In Admin	13
5	Allow ForeignKey Fields In Admin List Display	17
6	Django Migrations	21
6.1	Safe/Unsafe Migrations	21
6.2	Schema/Data Migrations	21
7	Log SQL Queries To Console	23
7.1	Using debug-toolbar	23
7.2	Using django-extensions	23

8	ORM Gotchas	27
8.1	Caching	27
8.2	Eager evaluation	27
8.3	Lazy evaluation	27
9	Form with multiple submit buttons	29
10	Dealing With CSRF Token Outside Of Django	31
10.1	Ajax	31
10.2	Postman	31
10.3	Shell	32
11	Laziness & Caching	35
12	Dynamic Initial Values In Forms	37
13	Management commands	41
14	Profiling & Optimizing Dango	43
14.1	What to Optimize?	43
14.2	Metrics	43
14.3	Aggregration	44
14.4	Profiling	45
14.5	Optimizing For Performance	45
15	Appendix A	47
15.1	Shell	47
15.2	iPython	48

PREFACE

1.1 Why this book?

blog posts

1.2 Who should read this book?

users

1.3 Acknowledgements

Krace Kumar

Haris Ibrahim

Tim Graham, <https://techytim.com/>

Andrew Godwin, <http://www.aeracode.org/>

Haki Banita

<https://hakibenita.com/>

AUTO REGISTER ALL MODELS IN ADMIN

2.1 Manual Registration

Inbuilt admin interface is one the most powerful & popular feature of Django. Once we create the models, we need to register them with admin, so that it can read schema and populate interface for it.

Let us register Book model in the admin interface.

```
# file: library/book/admin.py

from django.apps import apps

from book.models import Book

class BookAdmin(admin.ModelAdmin):
    list_display = ('id', 'name', 'author')

admin.site.register(Book, BookAdmin)
```

Now, we can see the book model in admin.

Action:

▼

Go

0 of 4 selected

<input type="checkbox"/>	ID	NAME	AUTHOR	IS AVAILABLE
<input type="checkbox"/>	10	Fluent Python	Luciano Ramalho	✖
<input type="checkbox"/>	3	Modern man in search of soul	C. J. Jung	✔
<input type="checkbox"/>	2	The Happines Hypothesis	Jonathan haidt	✖
<input type="checkbox"/>	1	1984	George orwell	✖

If the django project has too many models to be registered in admin or if it has a legacy database where all tables need to be registered in admin, then adding all those models to admin becomes a tedious task.

2.2 Auto Registration

To automate this process, we can programatically fetch all the models in the project and register them with admin. Also, we need to ignore models which are already registered with admin as django doesn't allow regsitering same model twice.

```
from django.apps import apps

models = apps.get_models()

for model in models:
    try:
        admin.site.register(model)
    except admin.sites.AlreadyRegistered:
        pass
```

This code snippet should run after all *admin.py* files are loaded so that auto registration happends after all manually added models are registered. Django provides `AppConfig.ready()` to perform any initialization tasks which can be used to hook this code.


```
# file: library/book/apps.py

from django.apps import apps, AppConfig
from django.contrib import admin

class BookAppConfig(AppConfig):

    def ready(self):
        models = apps.get_models()
        for model in models:
            try:
                admin.site.register(model)
            except admin.sites.AlreadyRegistered:
                pass
```

In the admin, we can see manually registered models and automatically registered models. If we open admin page for any auto registered model, it will show something like this.

Action: 0 of 5 selected

<input type="checkbox"/>	AUTHOR
<input type="checkbox"/>	Author object (6)
<input type="checkbox"/>	Author object (5)
<input type="checkbox"/>	Author object (4)
<input type="checkbox"/>	Author object (3)

This view is not at all useful for the users who want to see the data. It will be more informative if we can show all the fields of the model in admin.

2.3 Auto Registration With Fields

To achieve that, we can create an admin class to populate model fields in *list_display*. While registering, we can use this admin class to register the model.

```
from django.apps import apps, AppConfig
from django.contrib import admin

class ListModelAdmin(admin.ModelAdmin):
    def __init__(self, model, admin_site):
        self.list_display = [field.name for field_
→in model._meta.fields]
        super().__init__(model, admin_site)

class BookAppConfig(AppConfig):

    def ready(self):
        models = apps.get_models()
        for model in models:
            try:
                admin.site.register(model, _
→ListModelAdmin)
            except admin.sites.AlreadyRegistered:
                pass
```

Now, if we look at Author admin page, it will be shown with all relevant fields.

Action: 0 of 4 selected

<input type="checkbox"/>	ID	NAME	ACTIVE
<input type="checkbox"/>	6	Luciano Ramalho	✓
<input type="checkbox"/>	4	C. J. Jung	✗
<input type="checkbox"/>	3	Jonathan haidt	✗
<input type="checkbox"/>	2	George orwell	✓

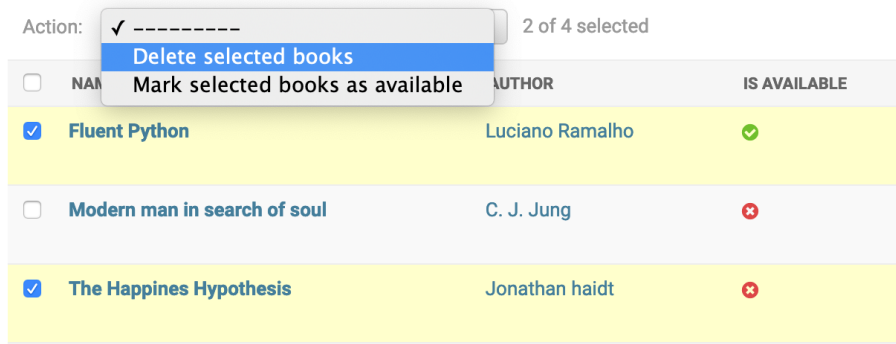
Since we have auto registration in place, when a new model is added or columns are altered for existing models, admin interface will update accordingly without any code changes.

CUSTOM ADMIN ACTIONS FOR QUERYSETS & INDIVIDUAL OBJECTS

3.1 Custom Actions On Querysets

Django provides admin actions which work on a queryset level. By default, django provides delete action in the admin.

In our books admin, we can select a bunch of books and delete them.



Django provides an option to hook user defined actions to run additional actions on selected items. Let us write a custom admin action to mark selected books as available.

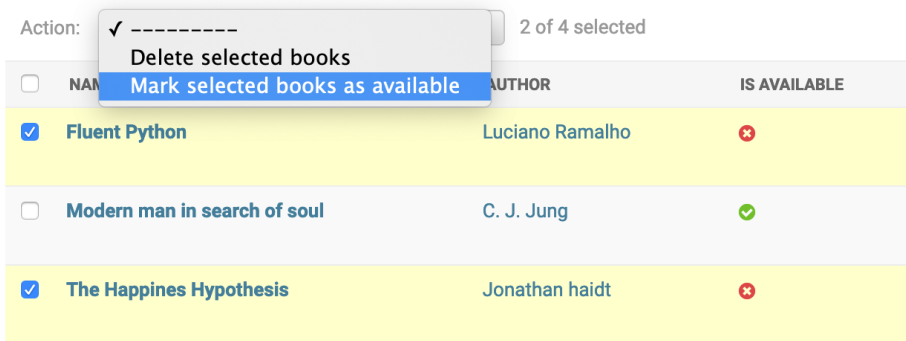
```
class BookAdmin(admin.ModelAdmin):  
    actions = ('make_books_available',)
```

(continues on next page)

(continued from previous page)

```
list_display = ('id', 'name', 'author')

def make_books_available(self, modeladmin,
    ↪request, queryset):
    queryset.update(is_available=True)
    make_books_available.short_description = "Mark_
    ↪selected books as available"
```



3.2 Custom Actions On Individual Objects

Custom admin actions are inefficient when taking action on an individual object. For example, to delete a single user, we need to follow these steps.

1. Select the checkbox of the object.
2. Click on the action dropdown.
3. Select “Delete selected” action.
4. Click on Go button.
5. Confirm that the objects needs to be deleted.

Just to delete a single record, we have to perform 5 clicks. That’s too many clicks for a single action.

To simplify the process, we can have delete button at row level. This

can be achieved by writing a function which will insert delete button for every record.

ModelAdmin instance provides a set of named URLs for CRUD operations. To get object url for a page, URL name will be `{{ app_label }}_{{ model_name }}_{{ page }}`.

For example, to get delete URL of a book object, we can call `reverse("admin:book_book_delete", args=[book_id])`. We can add a delete button with this link and add it to `list_display` so that delete button is available for individual objects.

```
from django.contrib import admin
from django.utils.html import format_html

from book.models import Book

class BookAdmin(admin.ModelAdmin):
    list_display = ('id', 'name', 'author', 'is_
↪available', 'delete')

    def delete(self, obj):
        view_name = "admin:{}_{}_delete".format(obj.
↪_meta.app_label, obj._meta.model_name)
        link = reverse(view_name, args=[obj.pk])
        html = '<input type="button" onclick=
↪"location.href=\'{}\'" value="Delete" />'.
↪format(link)
        return format_html(html)
```

Now in the admin interface, we have delete button for individual objects.

Action: 0 of 4 selected

<input type="checkbox"/>	ID	NAME	AUTHOR	IS AVAILABLE	DELETE
<input type="checkbox"/>	10	Fluent Python	Luciano Ramalho	✖	<input type="button" value="Delete"/>
<input type="checkbox"/>	3	Modern man in search of soul	C. J. Jung	✔	<input type="button" value="Delete"/>
<input type="checkbox"/>	2	The Happines Hypothesis	Jonathan haidt	✖	<input type="button" value="Delete"/>

To delete an object, just click on delete button and then confirm to delete it. Now, we are deleting objects with just 2 clicks.

In the above example, we have used an inbuilt model admin delete view. We can also write custom view and link those views for custom actions on individual objects. For example, we can add a button which will mark the book status to available.

In this chapter, we have seen how to write custom admin actions which work on single item as well as bulk items.

HYPERLINK FOREIGNKEYS TO ITS CHANGE VIEW IN ADMIN

Consider Book model which has Author as foreignkey.

```
from django.db import models

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

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.ForeignKey(Author)
```

We can register these models with admin interface as follows.

```
from django.contrib import admin

from .models import Author, Book

class BookAdmin(admin.ModelAdmin):
    list_display = ('name', 'author', )

admin.site.register(Author)
admin.site.register(Book, BookAdmin)
```

Once they are registered, admin page shows Book model like this.

Django administration

Home › Book › Books

Select book to change

Action: 0 of 2 selected

<input type="checkbox"/>	NAME	AUTHOR
<input type="checkbox"/>	Chalam	Chalam
<input type="checkbox"/>	The stranger	Albert Camus

2 books

While browsing books, we can see book name and author name. Here, book name field is linked to book change view. But author field is shown as plain text.

If we have to modify author name, we have to go back to authors admin page, search for relevant author and then change name.

This becomes tedious if users spend lot of time in admin for tasks like this. Instead, if author field is hyperlinked to author change view, we can directly go to that page and change the name.

Django provides an option to access admin views by its URL reversing system. For example, we can get change view of author model in book app using `reverse("admin:book_author_change", args=id)`. Now we can use this url to hyperlink author field in book admin.

```
from django.contrib import admin
from django.utils.safestring import mark_safe

class BookAdmin(admin.ModelAdmin):
    list_display = ('name', 'author_link', )

    def author_link(self, book):
        url = reverse("admin:book_author_change", _
↪args=[book.author.id])
```

(continues on next page)

(continued from previous page)

```
link = '<a href="%s">%s</a>' % (url, book.  
↪author.name)  
    return mark_safe(link)  
    author_link.short_description = 'Author'
```

Now in the book admin view, author field will be hyperlinked to its change view and we can visit just by clicking it.

Depending on requirements, we can link any field in django to other fields or add custom fields to improve productivity of users in admin.

Custom hyper links

[https://docs.djangoproject.com/en/dev/ref/models/instances/
#get-absolute-url](https://docs.djangoproject.com/en/dev/ref/models/instances/#get-absolute-url)

ALLOW FOREIGNKEY FIELDS IN ADMIN LIST DISPLAY

Django admin has *ModelAdmin* class which provides options and functionality for the models in admin interface. It has options like *list_display*, *list_filter*, *search_fields* to specify fields for corresponding actions.

search_fields, *list_filter* and other options allow to include a ForeignKey or ManyToMany field with lookup API follow notation. For example, to search by book name in Bestselleradmin, we can specify *book__name* in search fields.

```
from django.contrib import admin

from book.models import BestSeller

class BestSellerAdmin(RelatedFieldAdmin):
    search_fields = ('book__name', )
    list_display = ('id', 'year', 'rank', 'book')

admin.site.register(Bestseller, BestSellerAdmin)
```

However Django doesn't allow the same follow notation in *list_display*. To include ForeignKey field or ManyToMany field in the list display, we have to write a custom method and add this method in list display.

```
from django.contrib import admin

from book.models import Bestseller

class BestsellerAdmin(RelatedFieldAdmin):
    list_display = ('id', 'rank', 'year', 'book',
        ↳ 'author')
    search_fields = ('book__name', )

    def author(self, obj):
        return obj.book.author
    author.description = 'Author'

admin.site.register(Bestseller, BestsellerAdmin)
```

This way of adding foreignkeys in `list_display` becomes tedious when there are lots of models with foreignkey fields.

We can write a custom admin class to dynamically set the methods and attributes so that we can use the `ForeignKey` fields in `list_display`.

```
def get_related_field(name, admin_order_field=None,
    ↳ short_description=None):
    related_names = name.split('__')

    def dynamic_attribute(obj):
        for related_name in related_names:
            obj = getattr(obj, related_name)
        return obj

    dynamic_attribute.admin_order_field = admin_
    ↳ order_field or name
    dynamic_attribute.short_description = short_
    ↳ description or related_names[-1].title().replace(
    ↳ '_', ' ')
    return dynamic_attribute
```

(continues on next page)

(continued from previous page)

```
class RelatedFieldAdmin(admin.ModelAdmin):
    def __getattr__(self, attr):
        if '__' in attr:
            return get_related_field(attr)

        # not dynamic lookup, default behaviour
        return self.__getattribute__(attr)

class BestsellerAdmin(RelatedFieldAdmin):
    list_display = ('id', 'rank', 'year', 'book',
        ↪ 'book__author')
```

By subclassing `RelatedFieldAdmin`, we can directly use foreignkey fields in list display.

However, this will lead to N+1 problem. We will discuss more about this and how to fix this in orm optimizations chapter.

DJANGO MIGRATIONS

6.1 Safe/Unsafe Migrations

6.1.1 test

6.2 Schema/Data Migrations

Mixing schema and data migrations

set constraints

split migrations

LOG SQL QUERIES TO CONSOLE

Django ORM makes easy to interact with database. To understand what is happening behind the scenes or to see SQL performance, we can log all the SQL queries that are being executed. In this article, we will see various ways to achieve this.

7.1 Using debug-toolbar

Django debug toolbar provides panels to show debug information about requests. It has SQL panel which shows all executed SQL queries and time taken for them.

When building REST APIs or micro services where django templating engine is not used, this method won't work. In these situations, we have to log SQL queries to console.

7.2 Using django-extensions

Django-extensions provides lot of utilities for productive development. For `runserver_plus` and `shell_plus` commands, it accepts an optional `-print-sql` argument, which prints all the SQL queries that are being executed.

`./manage.py runserver_plus --print-sql` `./manage.py shell_plus --print-sql`
Whenever an SQL query gets executed, it prints the query and time taken for it in console.

In [42]: `User.objects.filter(is_staff=True)` Out[42]: `SELECT`
`“auth_user”.“id”,`

```
    “auth_user”.“password”,
    “auth_user”.“last_login”,
    “auth_user”.“is_superuser”,
    “auth_user”.“username”,
    “auth_user”.“first_name”,
    “auth_user”.“last_name”,
    “auth_user”.“email”,
    “auth_user”.“is_staff”,
    “auth_user”.“is_active”,
    “auth_user”.“date_joined”
```

`FROM “auth_user”`

`WHERE “auth_user”.“is_staff” = true LIMIT 21`

Execution time: 0.002107s [Database: default]

<QuerySet [<User: anand>, <User: chillar>]> Using django-querycount
Django-querycount provides a middleware to show SQL query count and show duplicate queries on console.

```
|-----|-----|-----|-----| | Type | Database |
Reads | Writes | Totals | Duplicates | |-----|-----|-----|
-----|-----| | RESP | default | 3 | 0 | 3 | 1 | |-----|
|-----|-----|-----| Total queries: 3 in 1.7738s
```

Repeated 1 times. `SELECT “django_session”.“session_key”,`
`“django_session”.“session_data”, “django_session”.“expire_date”`
`FROM “django_session” WHERE (“django_session”.“session_key”`
`= ‘dummy_key AND “django_session”.“expire_date” > ‘2018-05-`
`31T09:38:56.369469+00:00’:timestamptz)` This package provides
additional settings to customize output.

Django logging Instead of using any 3rd party package, we can use

django.db.backends logger to print all the SQL queries.

Add django.db.backends to loggers list and set log level and handlers.

```
'loggers': {  
    'django.db.backends': { 'level': 'DEBUG', 'handlers': ['console', ],  
},
```

In runserver console, we can see all SQL queries that are being executed.

```
(0.001)          SELECT          "django_admin_log"."id",  
"django_admin_log"."action_time",    "django_admin_log"."user_id",  
"django_admin_log"."content_type_id",  
"django_admin_log"."object_id",    "django_admin_log"."object_repr",  
"django_admin_log"."action_flag", "django_admin_log"."change_message",  
"auth_user"."id",    "auth_user"."password",    "auth_user"."last_login",  
"auth_user"."is_superuser",          "auth_user"."username",  
"auth_user"."first_name",          "auth_user"."last_name",  
"auth_user"."email",    "auth_user"."is_staff",    "auth_user"."is_active",  
"auth_user"."date_joined",          "django_content_type"."id",  
"django_content_type"."app_label",    "django_content_type"."model"  
FROM    "django_admin_log"  INNER JOIN    "auth_user"  
ON      ("django_admin_log"."user_id"    =    "auth_user"."id")  
LEFT    OUTER JOIN    "django_content_type"  ON  
("django_admin_log"."content_type_id" = "django_content_type"."id")  
WHERE   "django_admin_log"."user_id"    = 4 ORDER BY  
"django_admin_log"."action_time" DESC LIMIT 10; args=(4,)  
[2018/06/03 15:06:59] HTTP GET /admin/ 200 [1.69, 127.0.0.1:47734]  
These are few ways to log all SQL queries to console. We can also write  
a custom middleware for better logging of these queries and get some  
insights.
```


ORM GOTCHAS

N+1 Queries

8.1 Caching

8.2 Eager evaluation

8.3 Lazy evaluation

`book.author.id` `book.author_id`

`qs.exists()`

`.iterator()`

bulk operations

bulk update wont call save or signals Runpython wont call these

get only what you need

`values_list()`

enable query logging

FORM WITH MULTIPLE SUBMIT BUTTONS

submit1

submit2

DEALING WITH CSRF TOKEN OUTSIDE OF DJANGO

10.1 Ajax

When making ajax calls from browser, we need to set CSRF token.

10.2 Postman

Django has inbuilt CSRF protection mechanism for requests via unsafe methods to prevent Cross Site Request Forgeries. When CSRF protection is enabled on AJAX POST methods, X-CSRFToken header should be sent in the request.

Postman is one of the widely used tool for testing APIs. In this article, we will see how to set csrf token and update it automatically in Postman. CSRF Token In Postman

Django sets csrftoken cookie on login. After logging in, we can see the csrf token from cookies in the Postman.

We can grab this token and set it in headers manually.

But this token has to be manually changed when it expires. This process becomes tedious to do it on an expiration basis.

Instead, we can use Postman scripting feature to extract token from cookie and set it to an environment variable. In Test section of postman, add these lines.

```
var xsrfCookie = postman.getResponseCookie("csrftoken"); postman.setEnvironmentVariable('csrftoken', xsrfCookie.value);
```

This extracts csrf token and sets it to an environment variable called csrftoken in the current environment.

Now in our requests, we can use this variable to set the header.

When the token expires, we just need to login again and csrf token gets updated automatically. Conclusion

In this article we have seen how to set and renew csrftoken automatically in Postman. We can follow similar techniques on other API clients like CURL or httpie to set csrf token.

10.3 Shell

HTTPIe is an alternative to curl for interacting with web services from CLI. It provides a simple and intuitive interface and it is handy to send arbitrary HTTP requests while testing/debugging APIs.

When working with web applications that require authentication, using httpie is difficult as authentication mechanism will be different for different applications. httpie has in built support for basic & digest authentication.

To authenticate with Django apps, a user needs to make a GET request to login page. Django sends login form with a CSRF token. User can submit this form with valid credentials and a session will be initiated.

Establish session manually is boring and it gets tedious when working with multiple apps in multiple environments(development, staging, production).

I have written a plugin called httpie-django-auth which automates django authentication. It can be installed with pip

```
pip install httpie-django-auth
```

By default, it uses /admin/login to login. If you need to use some other URL for logging, set HTTPIE_DJANGO_AUTH_URL environment variable.

```
export HTTPIE_DJANGO_AUTH_URL='/accounts/login/'
```

Now you can send authenticated requests to any URL as

```
http :8000/profile -A=django --auth='username:password'
```

paas

iaas

ec2

home

**CHAPTER
ELEVEN**

LAZINESS & CACHING

DYNAMIC INITIAL VALUES IN FORMS

Django form fields accept initial argument. So You can set a default value for a field.

```
In [1]: from django import forms

In [2]: class SampleForm(forms.Form):
...:     name = forms.CharField(max_length=10,
↪ initial='avil page')
...:

In [3]: f = SampleForm()

In [4]: f.as_p()
Out[4]: u'<p>Name: <input maxlength="10" name="name"
↪ " type="text" value="avil page" /></p>'
```

Sometimes it is required to override init method in forms and set field initial arguments.

```
In [11]: from django import forms

In [12]: class AdvancedForm(forms.Form):
...:
...:     def __init__(self, *args, **kwargs):
...:         super().__init__(*args, **kwargs)
...:         self.fields['name'].initial =
↪ 'override'                                     (continues on next page)
```

(continued from previous page)

```
.....:
.....:         name=forms.CharField(max_length=10)
.....:

In [13]: f2 = AdvancedForm()

In [14]: f2.as_p()
Out[14]: '<p>Name: <input maxlength="10" name="name"
→" type="text" value="override" /></p>'
```

Now let's pass some initial data to form and see what happens.

```
In [11]: from django import forms

In [12]: class AdvancedForm(forms.Form):
.....:
.....:     def __init__(self, *args, **kwargs):
.....:         super().__init__(*args, **kwargs)
.....:         self.fields['name'].initial =
→'override' # don't try this at home
.....:
.....:         name=forms.CharField(max_length=10)
.....:

In [19]: f3 = AdvancedForm(initial={'name':
→'precedence'})

In [20]: f3.as_p()
Out[20]: '<p>Name: <input maxlength="10" name="name"
→" type="text" value="precedence" /></p>'
```

If You look at the value of input field, it's is NOT the overridden. It still has form initial value!

If You look into source code of django forms to find what is happening, You will find this.

```
data = self.field.bound_data(  
    self.data,  
    self.form.initial.get(self.name, self.field.  
↪initial)  # precedence matters!!!!  
)
```

So form's initial value has precedence over fields initial values.

So You have to override form's initial value instead of fields's initial value to make it work as expected.

```
In [21]: from django import forms  
  
In [22]: class AdvancedForm(forms.Form):  
    ....:  
    ....:     def __init__(self, *args, **kwargs):  
    ....:         super().__init__(*args, **kwargs)  
    ....:         self.initial['name'] = 'override'  
↪# aha!!!!  
    ....:  
    ....:         name=forms.CharField(max_length=10)  
    ....:  
  
In [23]: f4 = AdvancedForm(initial={'name':  
↪'precedence'})  
  
In [24]: f4.as_p()  
Out[24]: '<p>Name: <input maxlength="10" name="name"  
↪" type="text" value="override" /></p>'
```

Management command initial value

models initial value dict={ }

CHAPTER THIRTEEN

MANAGEMENT COMMANDS

ram .iterator()

cpu cron job frequent restarts

PROFILING & OPTIMIZING DANGO

14.1 What to Optimize?

When optimizing performance of web application, a common mistake is to start with optimizing the slowest page(or API) or going after micro optimizations which are not worth the effort.

In addition to considering response time, we should also consider the traffic it is receiving to prioritize the order of optimization.

14.2 Metrics

Let us profile our library django application and find performance bottlenecks.

```
pip install django-silk
```

Add silk to installed apps and include silk middleware in django settings.

```
MIDDLEWARE = [  
    ...  
    'silk.middleware.SilkyMiddleware',  
    ...  
]
```

(continues on next page)

(continued from previous page)

```
INSTALLED_APPS = (  
    ...  
    'silk'  
)
```

Run migrations so that Silk can create required database tables to store profile data.

```
$ python manage.py makemigrations  
$ python manage.py migrate  
$ python manage.py collectstatic
```

Include silk urls in root urlconf to view the profile data.

```
urlpatterns += [url(r'^silk/', include('silk.urls',  
    ↳ namespace='silk'))]
```

On silk requests page(<http://host/silk/requests/>), we can see all requests and sort them by overall time or time spent in database.

Silk creates `silk_request` table which contains information about the requests processed by django.

In this article, we learnt how to profile django webapp and identify bottlenecks to improve performance. In the next article, we will learn how to optimize these bottlenecks by taking an in-depth look at them.

14.3 Aggregation

Silk doesn't have support for aggregation of metrics and showing overall picture of the collected data on a dashboard.

We can write an admin view to show a dashboard of the metrics.

14.4 Profiling

`./manage.py runprofileserver`

14.5 Optimizing For Performance

APPENDIX A

15.1 Shell

zsh/fish

aliases

auto completion

Developers and hackers prefer using terminal and spend a lot of time on it. Instead of typing long commands over and over, they can be aliased to shortnames. The shell builtin alias allows users to set aliases.

One of the most used command while setting up development environment is `pip install -r requirements.txt`. This can be aliased to `pir`.

`alias pir='pip install -r requirements.txt'` Now to install requirements, type `pir` and pressing enter. Here are some other aliases related to python which will be useful on a daily basis.

```
alias py='python' alias ipy='ipython' alias py3='python3' alias  
ipy3='ipython3'
```

```
alias jn='jupyter notebook'
```

```
alias wo='workon' alias pf='pip freeze | sort' alias pi='pip install' alias  
pun='pip uninstall'
```

```
alias dj="python manage.py" alias drs="python manage.py run-  
server" alias drp="python manage.py runserverplus" alias dsh="python  
manage.py shell" alias dsp="python manage.py shell_plus" alias  
dsm="python manage.py schemamigration" alias dm="python man-  
age.py migrate" alias dmm="python manage.py makemigrations" alias  
ddd="python manage.py dumpdata" alias dld="python manage.py load-  
data" alias dt="python manage.py test" Just add the above aliases to your  
~/.bashrc or ~/.zshrc. That's it. Hpy alsng!
```

15.2 iPython

init file

auto reload