

The Django Admin Is Your Oyster



Let's Extend Its Functionality



by Adrienne Franke

- Can everybody hear me okay?
- First of all, I want to say thanks so much for coming out to this presentation
- I'm going to be sharing quite a bit about the Admin, how to customize it to your needs and some helpful tips and tricks
- I've posted the code and slides to my Github so don't feel like you have to take pictures of the slides
- The Django Admin Is Your Oyster, Let's Extend Its Functionality

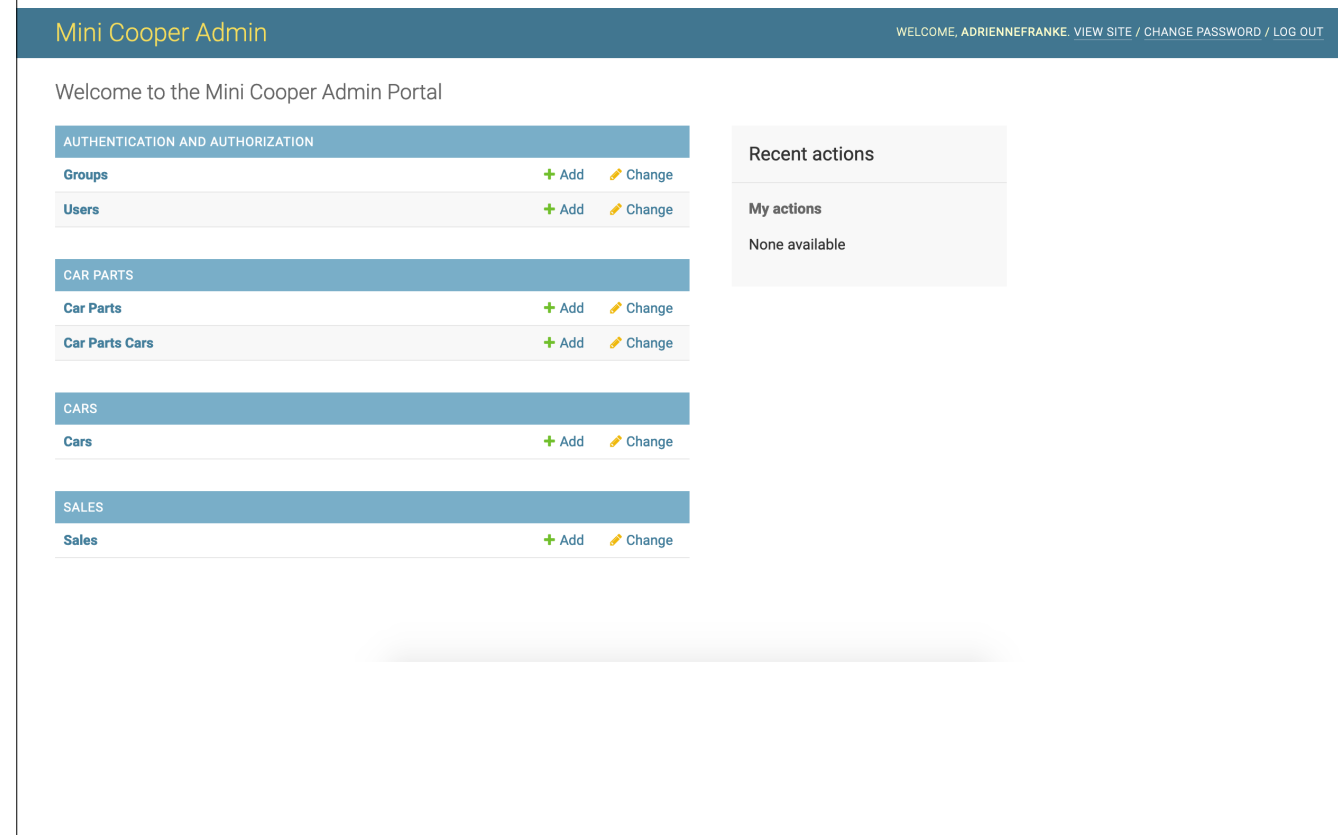
Who Am I?

- Software Engineer at 
- Live in Nashville, TN 
- Passionate about web development and automation
- Maintained a Django Admin app for a Data Science & Analytics department



- I'm Adrienne Franke - last name is pronounced "Frankie" even though it doesn't look like it
- I work in the healthcare space, I am a software engineer at Stratasean
- At my previous job, I built and maintained a Django Admin app for the entire Data Science & Analytics department
- That was where I learned all of these tips and tricks that I'm about to share with you

The Django Admin



- So how many people have used the Django Admin before?
- If you haven't used the Django Admin, this is what it looks like
- It's a GUI where users can interact with the contents of your database
- You've got the Authentication and Authorization section, which handles users and their permissions
- You can get very granular with your permissions here. Restrict some users from viewing certain models, allow only adds and deletes for certain models, it's very customizable.
- Then you've got your individual apps from your Django project and the different models you've defined and registered to appear here on this panel
- You can do all of the CRUD actions here: create, read, update, destroy rows in your database.

What It Is

- Internal tool for trusted users
- Powerful, out-of-the-box interface to manage data in database
- Highly customizable!

- The Django Admin is an interface where trusted users can manage the contents of your database
- It comes out of the box with Django and requires very little set up
- Django is one of the only frameworks that has something like this. Rails doesn't.
- The Django docs say “beware” of over customization. I say “Let's go for that!”

What It Isn't

- Front end for your end users
- Something to ignore

- While I think you can and should push the Django Admin to its limits, I do not think you should expose it as your front end for your end users
- The Django docs say this as well
- It's not secure and it's just not built for that
- While it's not an option for your front end, it is a very powerful tool, something that I don't think should be ignored
- It can empower administrators and trusted users across your organization - think Customer Success, Support, Analytics and Data Science in addition to your engineers



Speed Up Search Results



Posted by u/casual_drifter 6 years ago

14

Django Admin is really slow. Any thoughts on how to speed it up?



Django Admin Page with Inline Model Loads Very Slowly

Asked 5 years, 4 months ago Modified 4 years, 1 month ago Viewed 5k times



Posted by u/Teilchen 1 year ago

27

Django Admin Incredibly Slow Queries (~15 seconds) – Console takes 150ms

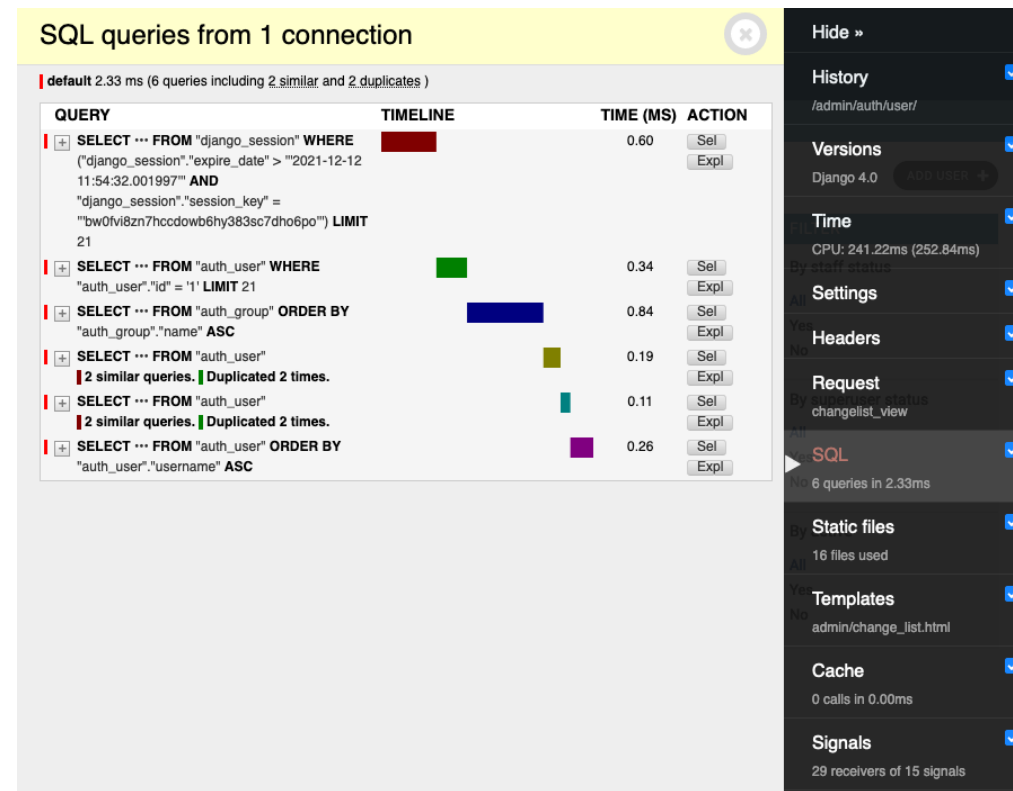


How to speed up Django admin page?

Asked 8 years, 8 months ago Modified 3 years, 9 months ago Viewed 4k times

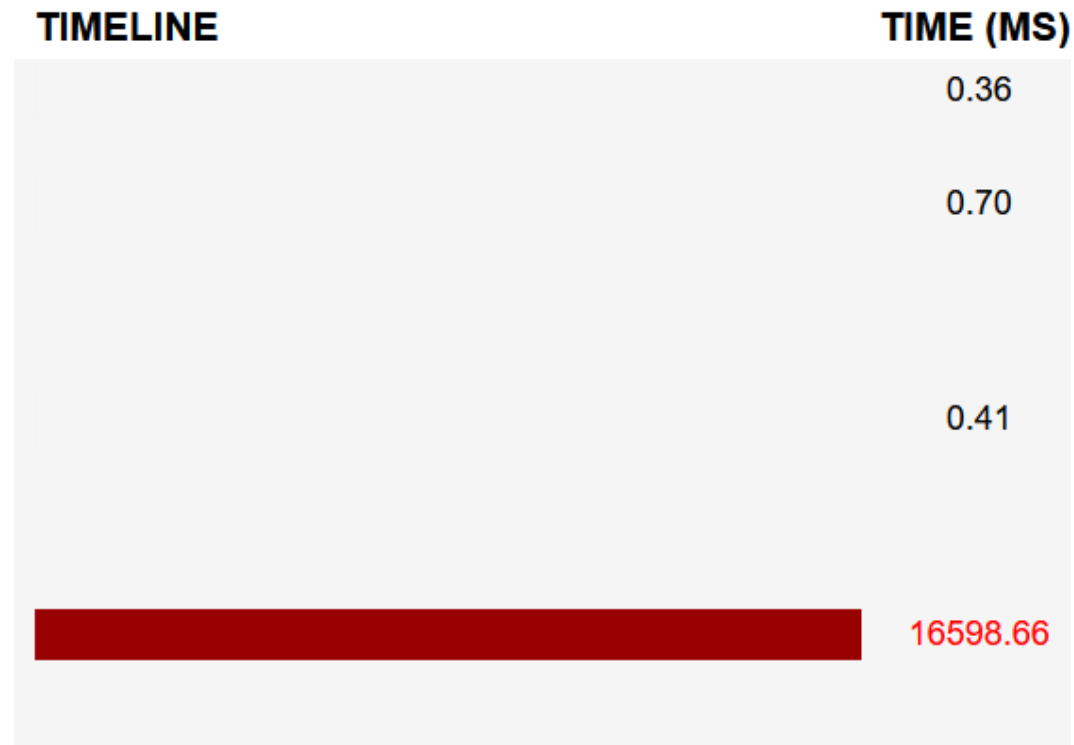
- The Django Admin is notorious for being slow, especially when you have a lot of data
- At my previous job, there are these two SQL tables
- One is 45 million rows and the other is 25 million rows
- Before optimizing, if you tried to search for some data in either of these tables, it would crash the entire app for everyone
- But I implemented a custom ``get_search_results`` function that now allows us to have lightning fast results without bringing down the app for these two models

Django Debug Toolbar



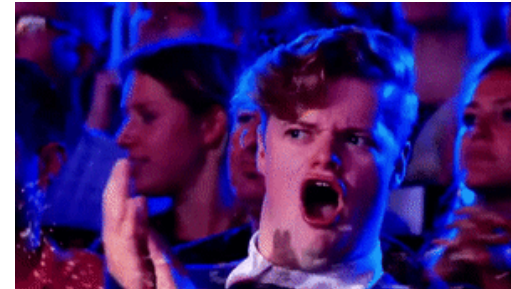
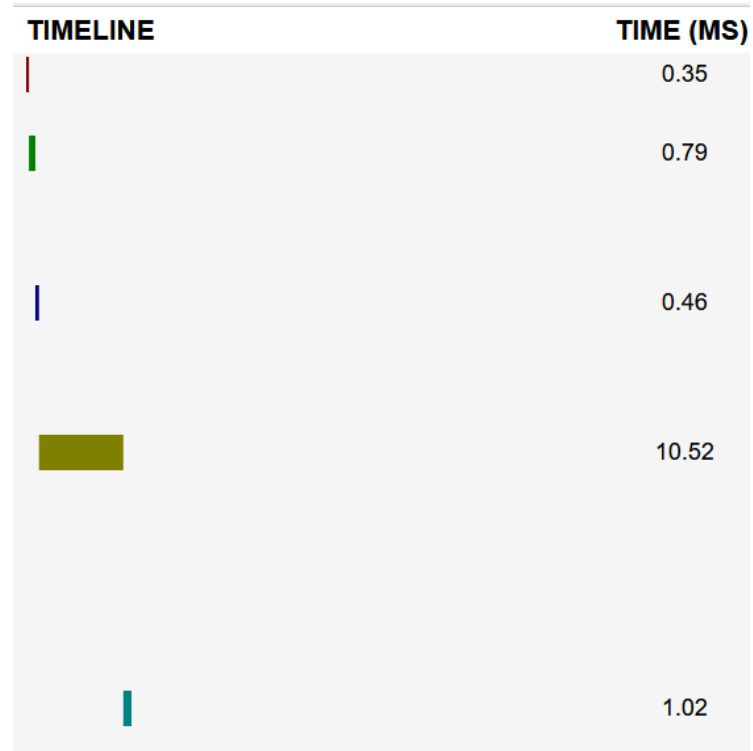
- The Django Debug Toolbar displays various debug data about the current request/response and you can kind of menu dive and get more detailed information
- It has info about time, headers, request, SQL, static files, templates, a bunch of great stuff
- This is what it looks like

Django Debug Toolbar



- I can't show the exact queries here but the Django Debug Toolbar shows you the exact SQL that's run when you click the search button and it tells you about how long each step takes
- I was able to see that it was pretty much querying every single related table for this model - which is a lot of data

Django Debug Toolbar



- After some optimization, I was able to take the query from 16 thousand milliseconds to 10 milliseconds



get_search_results()

```
def get_search_results(self, request, queryset, search_term):
    use_distinct = False
    if not search_term:
        return queryset, use_distinct
    else:
        return self.model.objects.filter(your filters here using search_term), use_distinct
```

- With just 6 lines of code, we can drastically speed up search results
- `get_search_results` is how Django returns the results based on your search query (or lack thereof)
- The reason this is faster is because you're only querying for the exact objects you're interested in, rather than every single related object
- Another thing you can do to speed this up is set `use_distinct = False`. Querying for distinct values is slower than not doing that.
- Main lesson here: if something is slow, use the Django Debug Toolbar to figure out where the bottlenecks are

prefetch_related()

- Automatically retrieves related objects and then does the joining in Python
- Reduces database queries

- prefetch_related() is another really handy function to use for speeding things up
- So this goes out and retrieves related objects in one batch and does all of the joining in Python
- It reduces database queries for you
- Docs: <https://docs.djangoproject.com/en/4.1/ref/models/querysets/#prefetch-related>

select_related()

- Creates a SQL JOIN and then includes the related objects in the SELECT statement
- Also reduces the amount of database queries

- `select_related()` is similar but it works differently
- It does it on the SQL side - it creates a SQL JOIN and includes related fields in the SELECT statement
- It also reduces database queries for you
- Docs: <https://docs.djangoproject.com/en/4.1/ref/models/querysets/#prefetch-related>

prefetch_related() or select_related()?

- prefetch_related(): group of things
- select_related(): ForeignKey and OneToOne relationships
- Try both and see how it goes

- Which one to use and when?
- prefetch_related() is for when you want to go get a group of things
- select_related() is limited to single objects - ForeignKey and OneToOne relationships
- But you can always just try both and see how it goes
- <https://stackoverflow.com/questions/31237042/whats-the-difference-between-select-related-and-prefetch-related-in-django-orm>

Behind the Scenes with Functions



- I'm going to talk a little bit about 2 functions that can be extremely useful for customizations
- I'll start with a little scenario
- Let's say that when you update a row in one table, you want to also update rows in another table, based on the updates you made in the first table
- For example, let's say you run a Mini Cooper factory and the price of a car part increases. You'd probably want to increase the price of the cars that use that car part.
- With the out-of-the-box Django Admin, there's no way to do that

Checkboxes

Change Car Part

HISTORY

Widget 31384

Part Name:

Widget 31384

Part Price:

2146.00



☐ Expand prices to cars with this part

Delete

Save and add another

Save and continue editing

SAVE

- Checkboxes offer a clever way to kick off functions that can do additional work for you
- You create them by using a form in your Django Admin page and defining a BooleanField
- If the box is checked, kick off a function on save
- You can do any type of pre or post save function here

clean()

```
class CarPartForm(forms.ModelForm):  
    expand_prices_to_cars_with_this_part = forms.BooleanField(required=False)  
  
    class Meta:  
        model = CarPart  
        fields = '__all__'  
  
    def clean(self):  
        cleaned_data = super().clean()  
  
        if cleaned_data.get('part_price') <= 0.00:  
            raise forms.ValidationError("Part price must be greater than $0.00")  
  
        self.cleaned_data['price_difference'] =  
            cleaned_data.get('part_price') -  
            CarPart.objects.filter(id=self.instance.id).values_list('part_price', flat=True)[0]  
        self.cleaned_data['cars_with_part'] =  
            CarPartCar.objects.filter(car_part=self.instance.id).values_list('car', flat=True)
```

- The clean() function can be used to check that the data in the form is valid
- For this example, we'll just make sure that all CarPart prices are greater than zero dollars
- The sky's the limit with the validations you can add in here - just define your logic and if the data doesn't hit what it needs to, raise a ValidationError
- You can also define objects here that you will use later in the save_model function
- For this example, we'll define two new objects: price_difference and cars_with_part
- Docs: <https://docs.djangoproject.com/en/4.1/ref/forms/validation/#cleaning-and-validating-fields-that-depend-on-each-other>

save_model()

```
@admin.register(CarPart)
class CarPartAdmin(admin.ModelAdmin):
    form = CarPartForm
    search_fields = ('part_name',)

    def save_model(self, request, obj, form, change):
        if form.cleaned_data.get('expand_prices_to_cars_with_this_part'):
            # insert function here that you want to happen when the box is checked

            car_part_id = obj.id
            price_difference = form.cleaned_data.get('price_difference')
            cars_with_part = CarPartCar.objects.filter(car_part=car_part_id).values_list('car', flat=True)

            for car in form.cleaned_data.get('cars_with_part'):
                old_msrp = Car.objects.filter(id=car).values_list('msrp', flat=True)[0]
                Car.objects.filter(id=car).update(msrp=old_msrp+price_difference)

        super().save_model(request, obj, form, change)
```

- The save_model function is how you can override the save function to do pre or post-save functions
- Here we are going to do something pre-save
- For every car that uses the part we're changing the price for, we are going to add the price difference to the MSRP price
- And that's only if we've checked that optional checkbox
- This will update the car's MSRP price and also the car part object as well
- Docs: https://docs.djangoproject.com/en/4.1/ref/contrib/admin/#django.contrib.admin.ModelAdmin.save_model

Multiple Databases, One Admin

- No out of the box support...
- But you can do it!

- There's no out of the box support for multiple databases in the Admin, but you can do it!
- Source: <https://docs.djangoproject.com/en/4.1/topics/db/multi-db/#database-routers>

Multiple Databases, One Admin

- Migrate using the `--database` option:
 - ``python manage.py migrate --database=sandbox``
- Set up custom database routing

- I'm not going to go into deep detail here on how to set this up because this talk is focused on the Admin
- But here are some tips
- By default, migrations happen on the 'default' database so when you want to migrate on the other databases you've defined you need to use the `--database` option
- Additionally, you need to set up custom database routing
- This custom database routing ensures that models stay "sticky" to their original database where they live
- You basically say "I want these groups of models to fall under this database always. So for these models, always read from the Sandbox database, always write to the Sandbox database"
- Source: <https://docs.djangoproject.com/en/4.1/topics/db/multi-db/#database-routers>

Multiple Databases, One Admin

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.sqlite3',  
        'NAME': BASE_DIR / 'db.sqlite3',  
    },  
    'sandbox': {  
        'ENGINE': 'django.db.backends.sqlite3',  
        'NAME': 'SANDBOX',  
    }  
}
```

- I will walk through how to get it running in the Admin though
- First, you'll need to adjust your settings.py file

Multiple Databases, One Admin

```
class SandboxCustomModelAdmin(admin.ModelAdmin):  
    """  
    Use this ModelAdmin when you want to use the 'sandbox' database  
    """  
    # A handy constant for the name of the alternate database  
    using = 'sandbox'  
  
    def save_model(self, request, obj, form, change):  
        # Tell Django to save objects to the 'sandbox' database  
        obj.save(using=self.using)  
  
    def delete_model(self, request, obj):  
        # Tell Django to delete objects from the 'sandbox' database  
        obj.delete(using=self.using)  
  
    def get_queryset(self, request):  
        # Tell Django to look for objects in the 'sandbox' database  
        return super().get_queryset(request).using(self.using)
```

- Next you'll need to define your own custom ModelAdmin
- You'll want to put this in a folder that's called 'utils' or 'common' since it might be used across your project
- You basically just tell the Django Admin to use the other database that you've defined in your settings.py file
- All of the models that use this database will still appear on your one Admin homepage
- There's no separate Admin link or anything
- Docs: <https://docs.djangoproject.com/en/4.1/topics/db/multi-db/>

Multiple Databases, One Admin

```
@admin.register(SandboxSale)
class SandboxSaleAdmin(SandboxCustomModelAdmin):
    list_display = ('car', 'sale_price', 'sale_date', )
    search_fields = ('sale_date', )
```

- This is how you'd register it in the Admin
- You just use the SandboxCustomModelAdmin class that you defined earlier

Multiple Databases, One Admin

Welcome to the Mini Cooper Admin Portal

AUTHENTICATION AND AUTHORIZATION

Groups [+ Add](#) [✎ Change](#)

Users [+ Add](#) [✎ Change](#)

CAR PARTS

Car Parts [+ Add](#) [✎ Change](#)

Car Parts Cars [+ Add](#) [✎ Change](#)

CARS

Cars [+ Add](#) [✎ Change](#)

SALES

Sales [+ Add](#) [✎ Change](#)

Sandbox Sales [+ Add](#) [✎ Change](#)

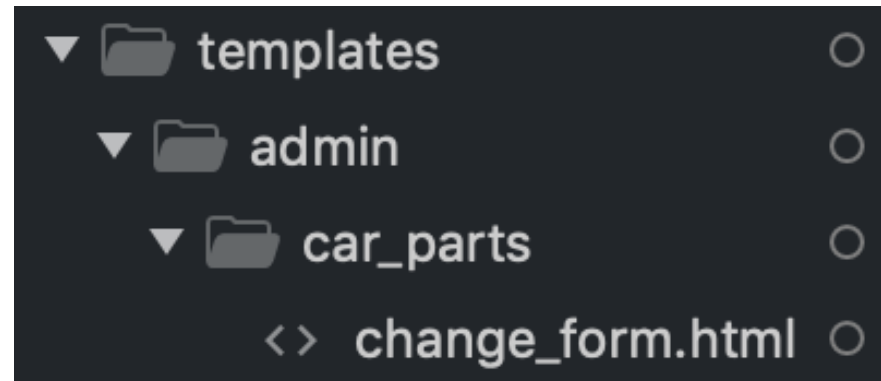
- Here it is!
- This can be really helpful if you're working with different databases
- Maybe different teams in your company use different databases
- This can be handy in those situations

Customize Admin Templates

```
TEMPLATES = [  
    {  
        'BACKEND': 'django.template.backends.django.DjangoTemplates',  
        'DIRS': [os.path.join(BASE_DIR, 'templates')],  
        'APP_DIRS': True,  
        'OPTIONS': {  
            'context_processors': [  
                'django.template.context_processors.debug',  
                'django.template.context_processors.request',  
                'django.contrib.auth.context_processors.auth',  
                'django.contrib.messages.context_processors.messages',  
            ],  
        },  
    },  
]
```

- Maybe you want to add a little more documentation to your Admin pages
- Something more broad than the `help_text` option for specific fields
- You'll need to start in your `settings.py` file and add the path to the `TEMPLATES` section

Customize Admin Templates



- Next, you'll want to create a folder structure like this

Customize Admin Templates

```
{% extends "admin/change_form.html" %}

{% block form_top %}
|     {{ original.admin_help_text }}
{% endblock %}
```

- This is where we'll specify where we want our specific documentation to appear on the change form

Customize Admin Templates

```
class CarPart(models.Model):
    part_name = models.CharField("Part Name", max_length=255, blank=False)
    part_price = models.DecimalField("Part Price", max_digits=8, decimal_places=2)

    admin_help_text = "Use this page to adjust the price of a car part."

    class Meta:
        verbose_name = "Car Part"

    def __str__(self):
        return f'{self.part_name}'
```

- On the model, we will define 'admin_help_text' and this is what will appear on the change_form

Customize Admin Templates

Change Car Part

HISTORY

Widget 42637

Use this page to adjust the price of a car part. You have the option of expanding the price to cars that use this car part as well.

Part Name:

Widget 42637

Part Price:

1538.00



☐ Expand prices to cars with this part

Delete

Save and add another

Save and continue editing

SAVE

- There's our Admin help text!
- Source: <https://stackoverflow.com/questions/6583877/how-to-override-and-extend-basic-django-admin-templates/29997719#29997719>

Dynamic Help Text

Change Car Part

Widget 42637

Use this page to adjust the price of a car part.

Part Name:

Widget 42637

Part Price:

1538.00



☐ Expand prices to cars with this part

Cars with this CarPart: Mini Coupe, Mini Paceman, Mini Countryman, Mini Cooper S

Delete

- Remember those checkboxes we were talking about earlier? What if we want to give that Checkbox a bit more context? What if we want to make it more user friendly?
- What if we want to change the help text for a field depending on the object itself? Or maybe depending on what the given object is related to?
- We can do that!

Dynamic Help Text

```
class CarPartForm(forms.ModelForm):  
    def __init__(self, *args, **kwargs):  
        super(CarPartForm, self).__init__(*args, **kwargs)
```

- The code is too much to put up on this screen all at once so I'll have to go section by section
- Basically what you do is your override the init function for your Form
- This is run before the form is shown to your users, it's the "initialize" function.
- So the first thing to do in this function is use the super() function. This function gives you access to the methods and properties of your CarPartForm

Dynamic Help Text

```
car_part_id = kwargs.get("instance").id
```

- Here we use kwargs to get the instance's id
- Kwargs are keyword arguments - they were passed in to the init function
- This is the object that you clicked to go to the change page
- For this example, it's the id of the CarPart object
- Once you have the id here, you can basically do whatever logic you want after that

Dynamic Help Text

- Then we'll get the related objects that we need
- And use the related objects to dynamically set our help text

```
self.fields['expand_prices_to_cars_with_this_part'].help_text =  
"Cars with this CarPart: {}".format(', '.join(list(cars_with_car_part)))
```

- Again, I'm not going to put a bunch of code up here because that wouldn't be helpful to you
- We'll use the ORM to get related objects that we need, like what are the cars that use these parts and what are those cars' names
- And then set our specific field's help text to the specific text for this specific model

Dynamic Help Text

- Error handling!

☐ Expand prices to cars with this part

There are no Cars that utilize this CarPart.

- Don't forget about error handling with this
- If your object doesn't have related objects, you'll want to say that in the help_text

Dynamic Help Text

Change Car Part

Widget 42637

Use this page to adjust the price of a car part.

Part Name:

Widget 42637

Part Price:

1538.00



☐ Expand prices to cars with this part

Cars with this CarPart: Mini Coupe, Mini Paceman, Mini Countryman, Mini Cooper S

Delete

- Here it is!
- A lot more clarity around what happens when I actually change a car part price with that checkbox selected

Custom Admin Actions

Action: ✓ ----- 0 of 100 selected

☐ MODEL NAME

☐ Mini Cooper S

☐ Mini JCW

☐ Mini Paceman

Delete selected cars

- Out of the box, there's just one Admin Action available and it's 'Delete'
- You can check as many rows as you want and go ahead and delete them
- But I bet there are a lot more things you'd like to be able to do
- For example, with my Mini Cooper factory, maybe I want to apply a discount to Mini Cooper if I'm running a sale for my customers
- Maybe I want to hit an external API and get some updated prices for my parts
- So I'm going to show you how to do that

Custom Admin Actions

```
def increase_msrp_by_8_perc(modeladmin, request, queryset):  
    for car in queryset:  
        car.msrp = car.msrp * decimal.Decimal('1.08')  
        car.save()  
increase_msrp_by_8_perc.short_description = 'Increase MSRP by 8%%'  
  
@admin.register(Car)  
class CarAdmin(admin.ModelAdmin):  
    list_display = ('model_name', 'year_released', 'msrp')  
    search_fields = ('model_name', 'year_released')  
    actions = [increase_msrp_by_8_perc, ]
```

- Let's take the example that we need to increase our prices because inflation has gone up a bit
- So first we need to define the function that we want to happen when we select our custom action and hit "Go"
- So for each car that we've selected, set the MSRP to be 8% higher than it was and save it
- The "short_description" that we define is what will show in the drop down so make it very clear what's going to happen
- Then we list the action under our Admin section
- Source: <https://simpleisbetterthancomplex.com/tutorial/2017/03/14/how-to-create-django-admin-list-actions.html>

Custom Admin Actions

Action: ✓ ----- 2 of 100 selected

<input type="checkbox"/>	MO
<input checked="" type="checkbox"/>	Mini Cooper S
<input checked="" type="checkbox"/>	Mini JCW
<input type="checkbox"/>	Mini Paceman
<input type="checkbox"/>	Mini Cooper S Clubman

- And there we have it!
- We can select whatever cars we want, select the “Increase MSRP by 8%” action and hit Go and our data will be updated!

To Recap...

- The Django Admin can be a great tool for your trusted, internal users
- Don't be scared of customizing it - in fact, go for it!
- Be creative in your endeavors 🎨
- Helpful resources: Django Debug Toolbar, Django Docs, StackOverflow (of course)

- Want to re-emphasize the “trusted, internal users” part here
- But otherwise, don't be scared of customizing the Django Admin!
- It can be really awesome and fast
- Lots of good info online to help you through tricky situations
- Just remember, the Django Admin is Your Oyster

Questions?

- Follow me on Twitter: @adriennefranke
- Read my blog: adriennefranke.com
- Email me: hi@adriennefranke.com
- Code + Slides: <https://github.com/adriennefranke/djangocon2022>