Django pt. 3: Models, Migrations, and CRUD

Kianoosh Abbasi

CSC309 Winter 2022

So far

- How web works, HTML, CSS
- Django intro
  Setup, simple views, forms, templates
- MVC Design patterns
- Working with a database
  ORM and models
- Authentication
  User model, sessions, login

This session

- Custom models

  Fields, relations, queries

- Migrations

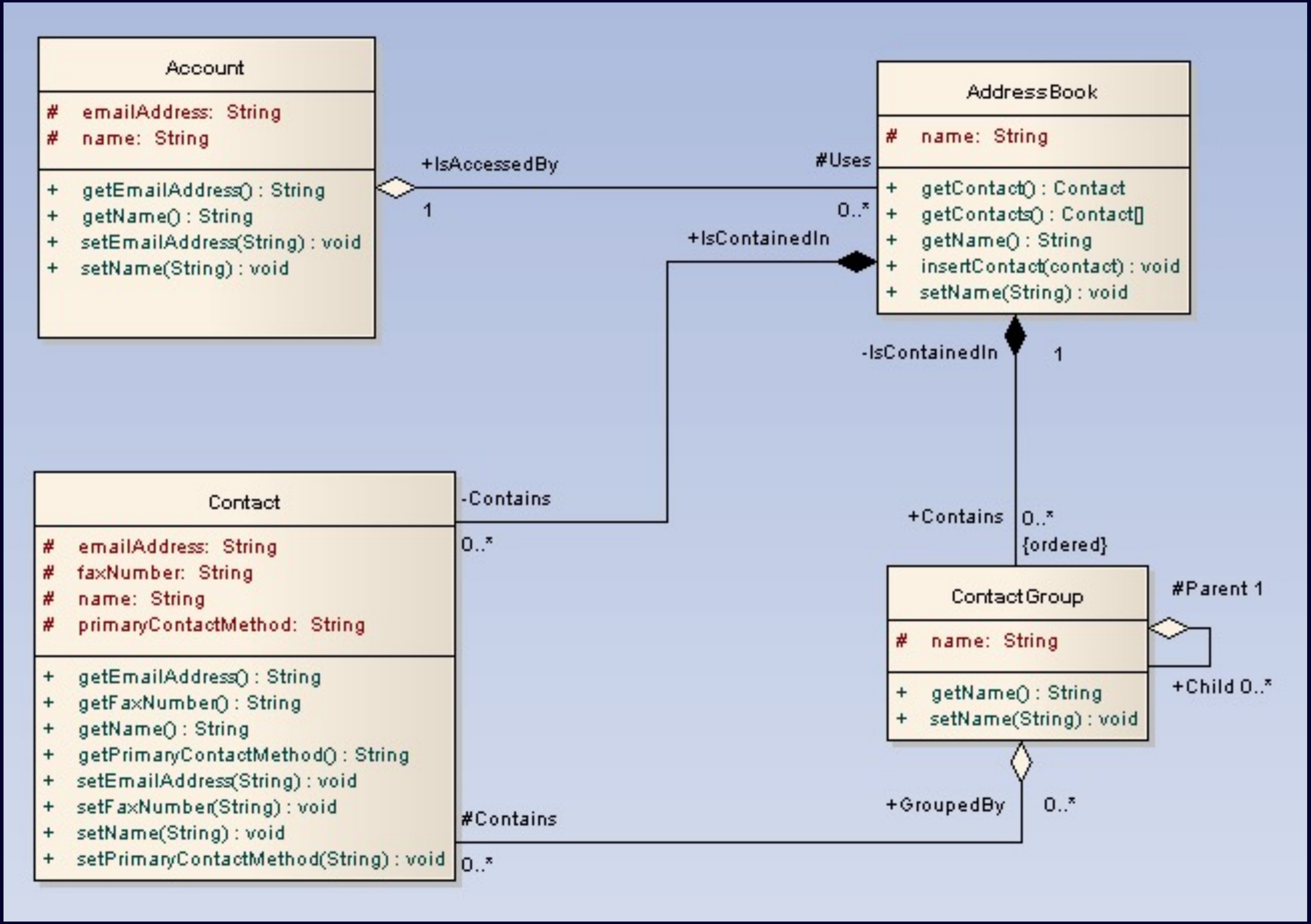
- Advanced views

  Class-based and CRUD views

Creating models

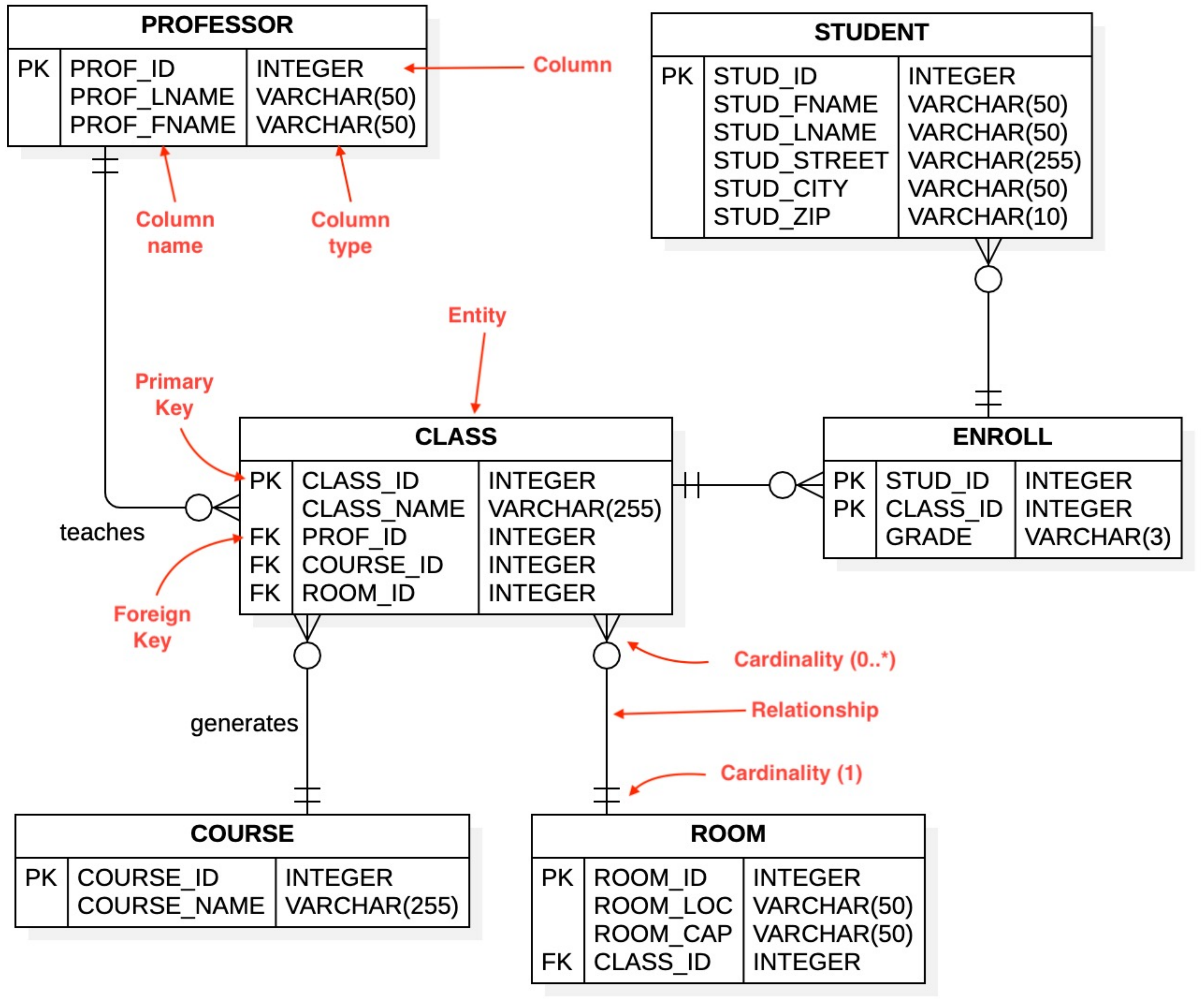
- MUST be done before coding starts
- Independent of programming language and framework
- Changing the models is not always easy
  - Especially in the production phase
- Models involve user data: the most sensitive part of your application
  - It's important to design secure and efficient models

Class diagram



Source: https://sparxsystems.com/images/screenshots/uml2_tutorial/cl01.png

ER diagram



Source: https://docs.staruml.io/working-with-additional-diagrams/entity-relationship-diagram

Creating models in Django

- Example: an online shopping application

- Potential models: user, store, product, order, shipment, etc.

- Think about Django apps

  Don’t forget to add your apps to `INSTALLED_APPS` in *settings.py*

Django models

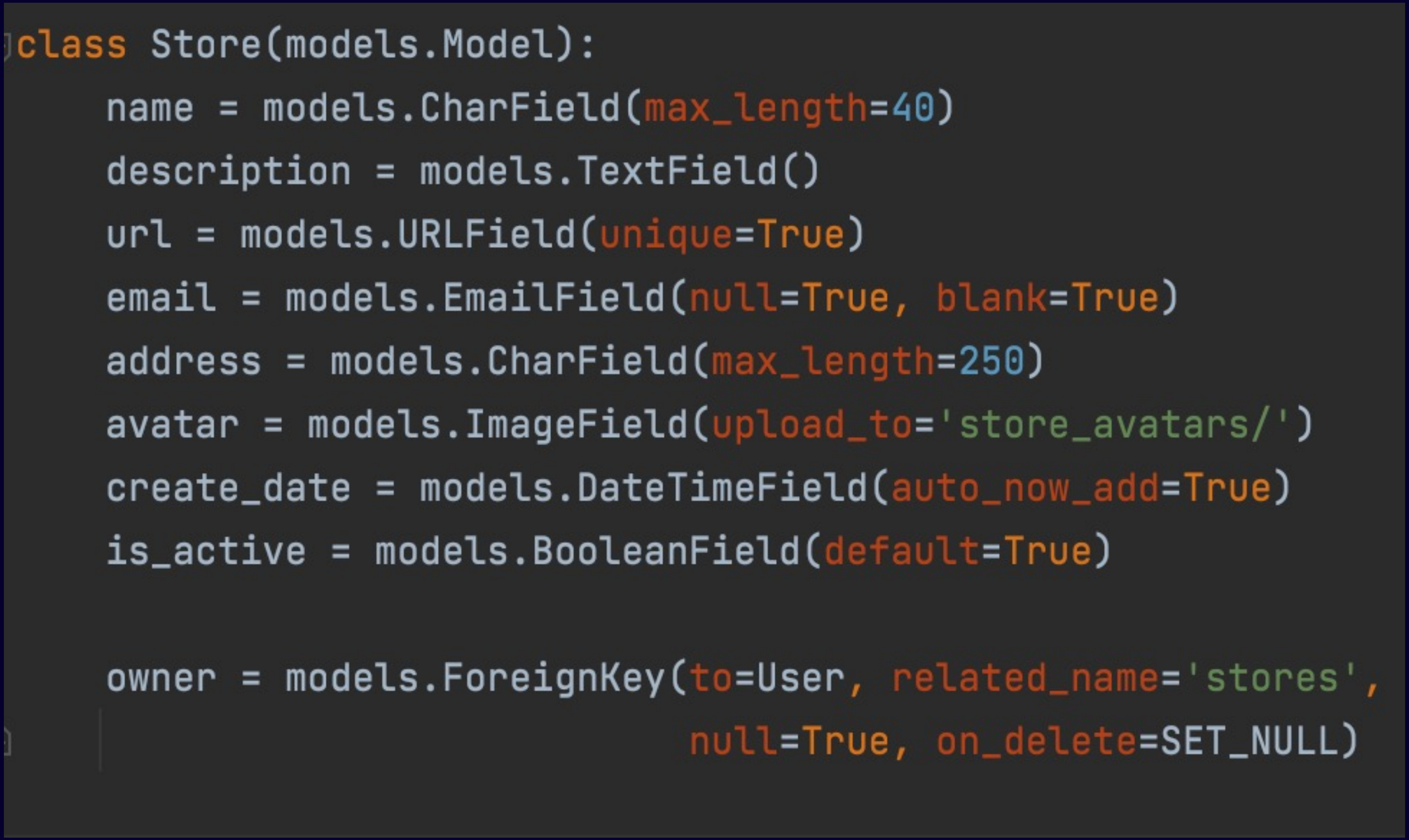
- Must be a subclass of `django.db.models.Model`
  - Pre-imported at the beginning of models.py
- Standard for big projects: create a models directory
  - Put each model in a separate file under that directory
- Add fields from the ER (or class) diagram to your model
  - Subclass of `django.db.models.Field`
  - Mapped to database column types by the ORM

Fields

Visit https://www.geeksforgeeks.org/django-model-data-types-and-fields-list/

- CharField
  - EmailField
  - URLField
  - TextField
- BooleanField
- IntegerField
  - AutoField
  - BigIntegerField
  - SmallIntegerField
- FloatField
  - DecimalField
- TimeField
  - DateField
  - DateTimeField
- FileField
  - ImageField

Example model



```
class Store(models.Model):
    name = models.CharField(max_length=40)
    description = models.TextField()
    url = models.URLField(unique=True)
    email = models.EmailField(null=True, blank=True)
    address = models.CharField(max_length=250)
    avatar = models.ImageField(upload_to='store_avatars/')
    create_date = models.DateTimeField(auto_now_add=True)
    is_active = models.BooleanField(default=True)

    owner = models.ForeignKey(to=User, related_name='stores',
                              null=True, on_delete=SET_NULL)
```

Making it work

- Every time your model changes, create and run migrations
```
./manage.py makemigrations
./manage.py migrate
```
- Register your model to the admin panel
  In admin.py: `admin.site.register(Store)`
- Custom admins can be created
- More on migrations and admin panel later this session

Null vs blank

- The null argument is attributed to database null condition

- Blank checks if the value submitted by forms (or the admin panel) is empty or not
  - Has no database effect

- Usually, both are either false (default) or true
  - Exceptional cases like SET_NULL in foreign key

More notes

- URLField and EmailField are just variations of CharField with a validator
  - Stored as VARCHAR in database
- Unlike CharField, TextField does not require a `max_length`
- Django automatically creates an AutoField named id
  - Used as the primary key

Other options

- Database options

  `unique=True` and `db_index=True`

- Admin panel and forms

  `help_text` and `verbose_name`

- Another column can be chosen as primary key

  `primary_key=True` (not a very good idea)

Foreign key

- Used for many-to-one and one-to-many relations
  - Defined at the foreign key end
  - Only stores the primary key in the database column
- Reverse traversal done by a field with the name defined by related_name
  - Default is <model_name>_set. E.g., `user.store_set.filter(…)`
- Foreign keys entail a separate query to fetch the related object(s)

Other relational fields

- OneToOneField

  A foreign column in database with `unique=True`

- ORM's reverse traversal returns a single object

  Default is <model_name> E.g., `user.store.name`

- ManyToManyField

  Easy interface that each side has a queryset of related objects

  A separate table in the database

File uploads

- Just file's path is saved in the database

- By default, the `upload_to` folder is created at project directory

  Not a good practice

- At settings, create a media root to gather all uploads

  ```
  MEDIA_ROOT = BASE_DIR / "media"
  ```

File uploads

- To access the file, a separate request is sent by the browser

  Translated to a file access by Django

- At settings, create a media URL to group all URL

  ```
  MEDIA_URL = "media/"
  ```

- Append the following array to the core urlpatterns

  ```
  static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
  ```

ORM functions

- Similar to those of User

- Examples:

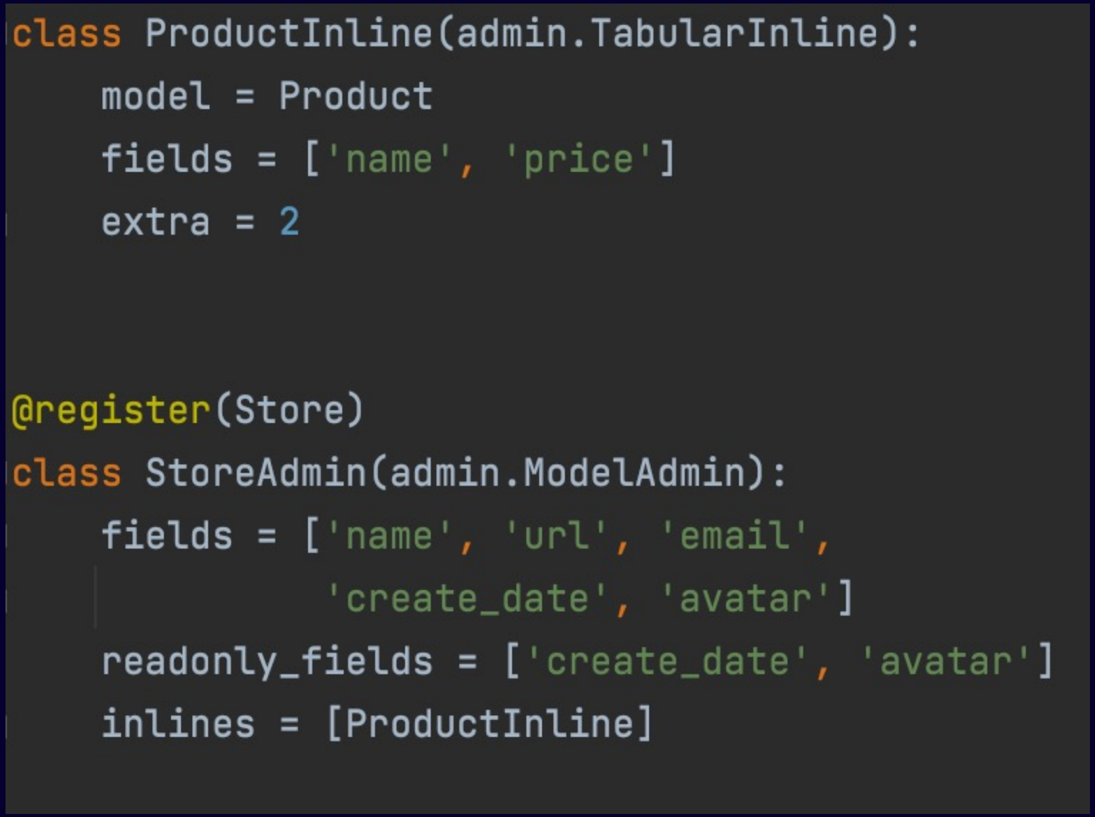
```
Store.objects.create(name='Apple', url='apple.com')
apple = Store.objects.filter(name__contains='Apple').first()
user = User.objects.get(username='test')
user.stores.add(apple)

apple.refresh_from_db()
apple.owner.first_name = 'Tim'
apple.owner.save()
```

Advanced admin panel (optional)

- Exact fields (editable and read-only) can be specified

- Use inlines to list all related objects

  Example: Product is a model with foreign key to Store



```
class ProductInline(admin.TabularInline):
    model = Product
    fields = ['name', 'price']
    extra = 2


@register(Store)
class StoreAdmin(admin.ModelAdmin):
    fields = ['name', 'url', 'email',
              'create_date', 'avatar']
    readonly_fields = ['create_date', 'avatar']
    inlines = [ProductInline]
```

Migrations

The great assumption

- The state of database tables is the same as what defined in model classes

- But these two are totally independent things
  Python classes vs database tables

- ORM’s job to apply application’s schema to database
  Via DDL queries

- Changes to schema's state:
  - Creation or removal of a table/model
  - Creation or removal of a column/field
  - Modification of field option/attributes

- Whenever the state changes, database should migrate to the new state

- Django does not do it automatically. WHY?

- In fact, Django does not even monitor the state change!
  - You simply get a database exception if ORM's and database's schema do not match

Migrations

- Think about it as a git commit

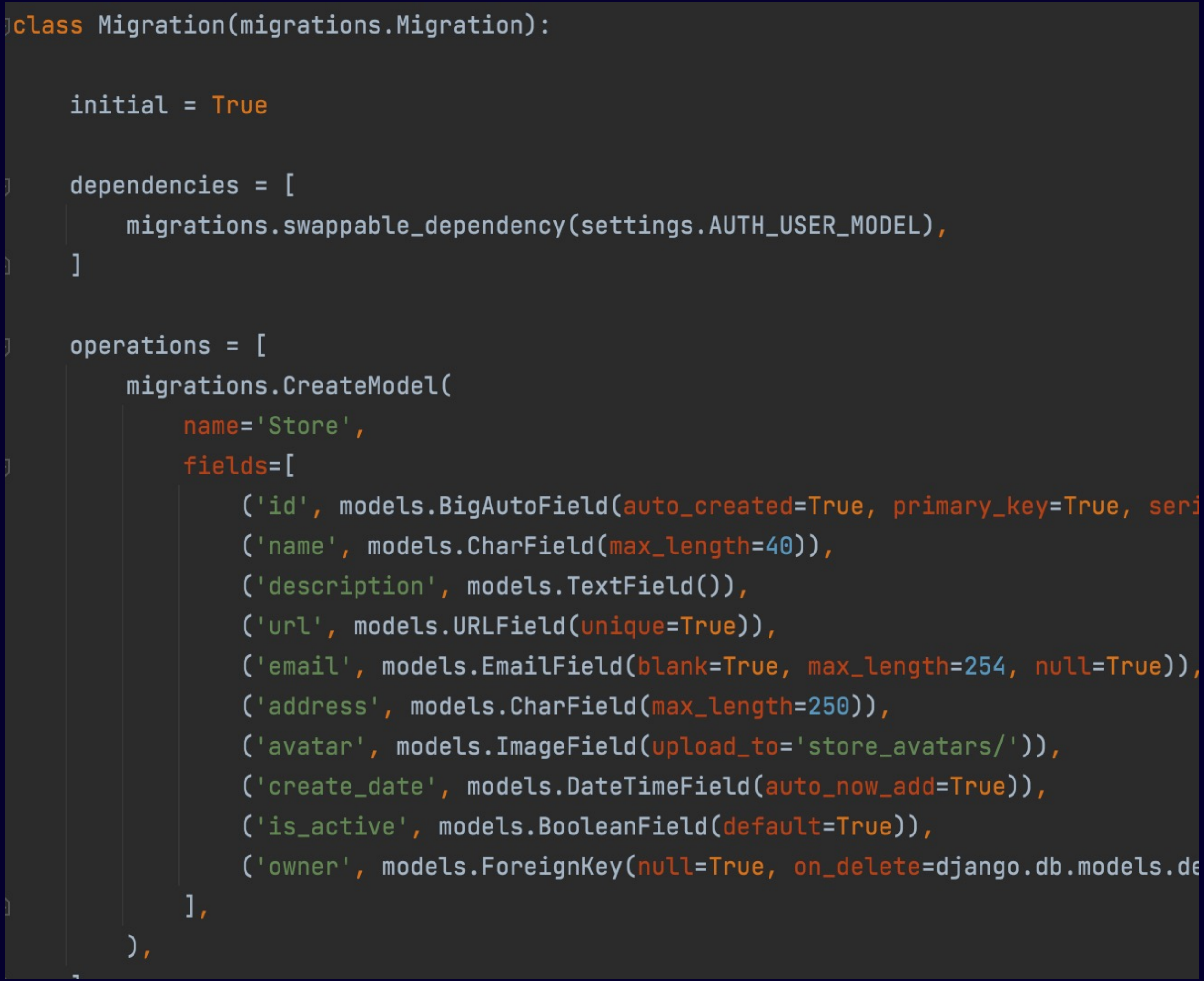
  Talks about what has changed since the last migration

- History of changes needs to be stored somewhere

- The migrations folder inside each app

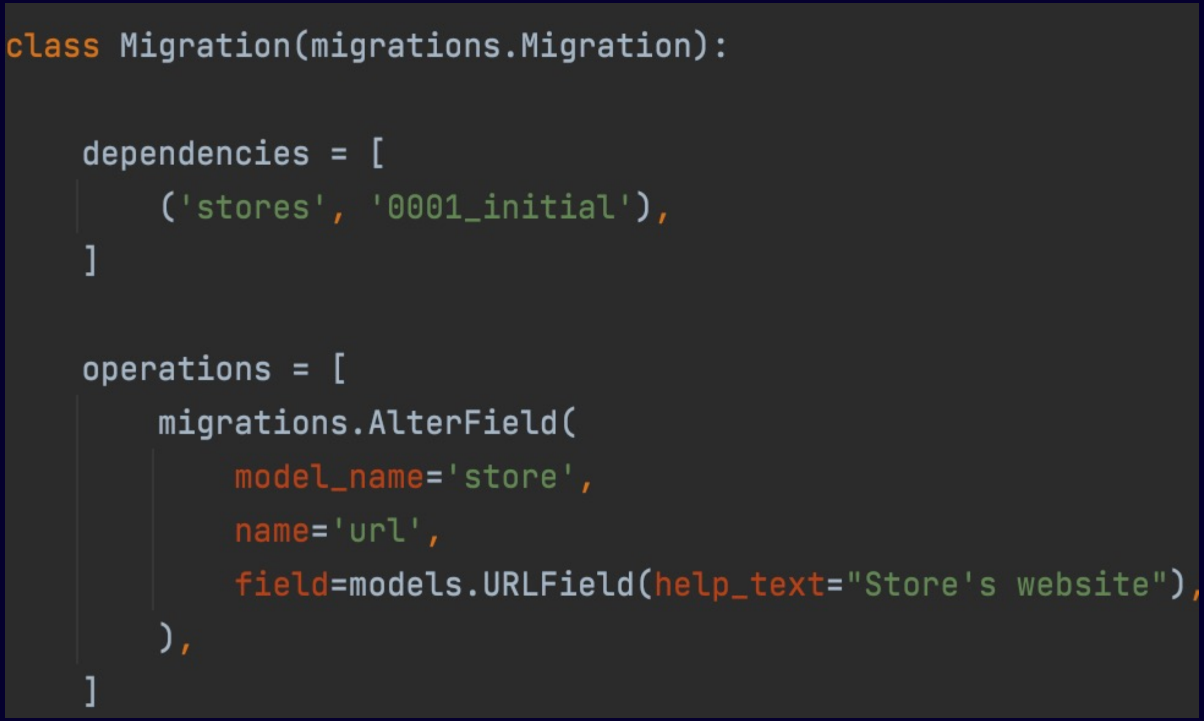
- Migrations are generated via `./manage.py makemigrations`

0001_initial



```
class Migration(migrations.Migration):

    initial = True

    dependencies = [
        migrations.swappable_dependency(settings.AUTH_USER_MODEL),
    ]

    operations = [
        migrations.CreateModel(
            name='Store',
            fields=[
                ('id', models.BigAutoField(auto_created=True, primary_key=True, seri
                ('name', models.CharField(max_length=40)),
                ('description', models.TextField()),
                ('url', models.URLField(unique=True)),
                ('email', models.EmailField(blank=True, max_length=254, null=True)),
                ('address', models.CharField(max_length=250)),
                ('avatar', models.ImageField(upload_to='store_avatars/')),
                ('create_date', models.DateTimeField(auto_now_add=True)),
                ('is_active', models.BooleanField(default=True)),
                ('owner', models.ForeignKey(null=True, on_delete=django.db.models.de
            ],
        ),
```

0002_alter_store_url



```
class Migration(migrations.Migration):

    dependencies = [
        ('stores', '0001_initial'),
    ]

    operations = [
        migrations.AlterField(
            model_name='store',
            name='url',
            field=models.URLField(help_text="Store's website"),
        ),
    ]
```

Makemigrations

- Builds a local model state from previous migrations
- But does not do ANY database operation to check that schema
  - Not its concern!
- Iterates over all Model's subclasses to find out differences
- Creates a new migration file for the corresponding apps

Applying the migrations

- DDL queries extracted from each migration file
- Applied to the database via `./manage.py migrate`
  App or migration name can be specified as well
- But a migration should not be applied twice!
  How is Django to know?
- Migrations themselves are stored in database

Applying the migrations

- Applied migrations stored in `django_migrations` table
- Only includes metadata (name, app, applied time)
  Content is only stored in the file
- The `migrate` command only applies those that are not present in that table

- Ideally, you never need to manipulate migration files/table

- Django migrations are like pointers in C
  - Powerful but dangerous!

- Does not enforce many of its underlying assumptions

- Migration errors can take hours to resolve
  - Don't mess with them unless you know what you are doing!

Migration errors

- A common scenario:
  You and your teammate add migrations independently
- Not always a problem: Migrations have dependencies
  Works like a git merge
- Otherwise, you can unapply or fake a migration

Unapply a migration

- Via `./manage.py migrate <app> <last_migration_name>`
  Or `./manage.py migrate <app> zero` to unapply all

- Rolls back its changes
  `CREATE TABLE -> DROP TABLE` (all its data is permanently lost!)
  `ALTER COLUMN -> ALTER COLUMN` (to its previous state)
  `DROP COLUMN -> ADD COLUMN` (to its previous state)

- The corresponding row is deleted from migration table
  The migration file can be safely deleted now!

- Never ever delete a migration file before it is unapplied!

Fake a migration (optional)

- Via `./manage.py migrate --fake`
- Only creates the database row for the migration
  Without actually executing the queries
- Use case: if the state of database is already ok, but there are unapplied migrations for some reason

The last resort

- Deleting the whole db.sqlite3 file clears up everything!
- Then you can delete all migration files and start over
- Definitely NOT an option in production
  - So be careful about migrations

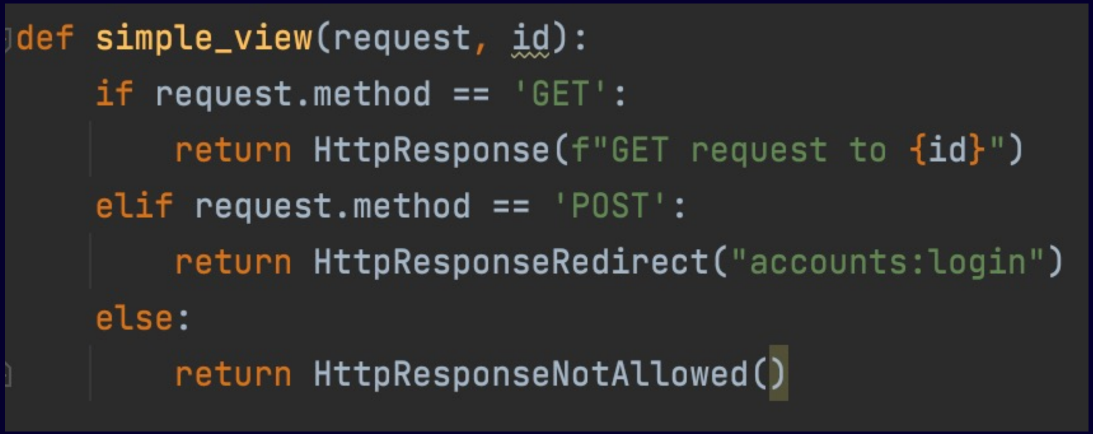
Advanced Views

Class-based (generic) views

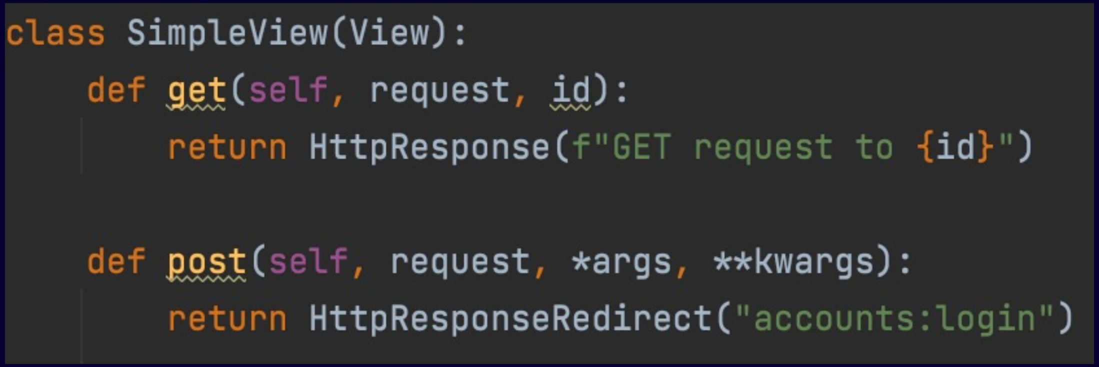
- Views can be numerous and big
- Class-based views: standard for medium/big projects
  A new instance created at every request: NO shared self object
- Create views directory and have each view in a separate file
- Subclass `django.views.View` and create a method (function) for each HTTP method

Function based

Class based

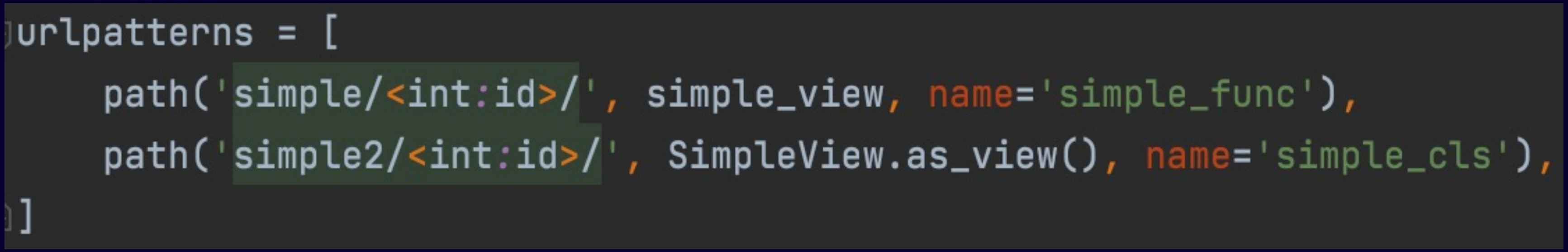


```
def simple_view(request, id):
    if request.method == 'GET':
        return HttpResponse(f"GET request to {id}")
    elif request.method == 'POST':
        return HttpResponseRedirect("accounts:login")
    else:
        return HttpResponseNotAllowed()
```



```
class SimpleView(View):
    def get(self, request, id):
        return HttpResponse(f"GET request to {id}")

    def post(self, request, *args, **kwargs):
        return HttpResponseRedirect("accounts:login")
```

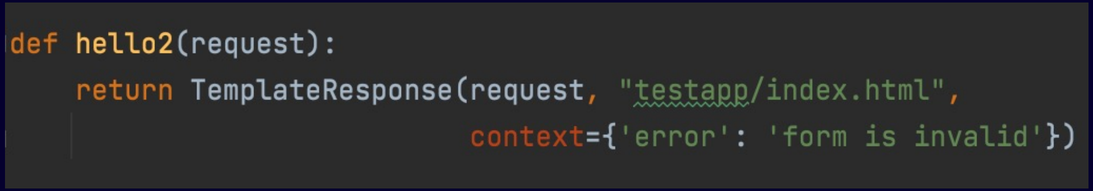
urls



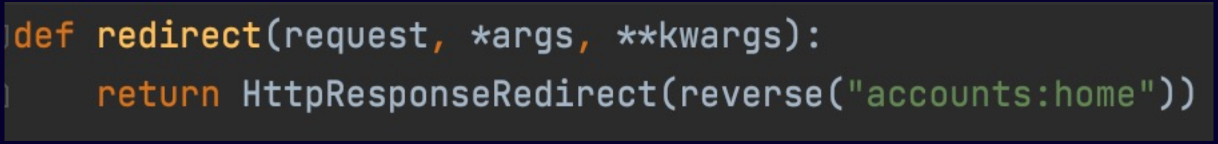
```
urlpatterns = [
    path('simple/<int:id>/', simple_view, name='simple_func'),
    path('simple2/<int:id>/', SimpleView.as_view(), name='simple_cls'),
]
```

TemplateView

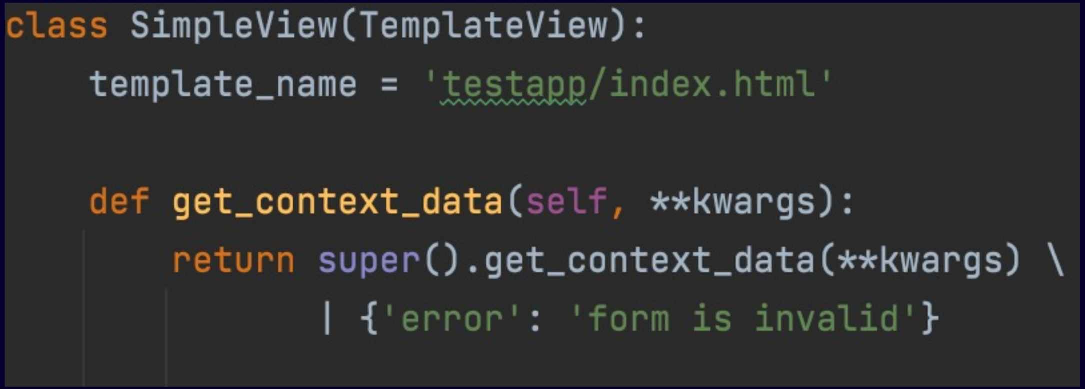
RedirectView



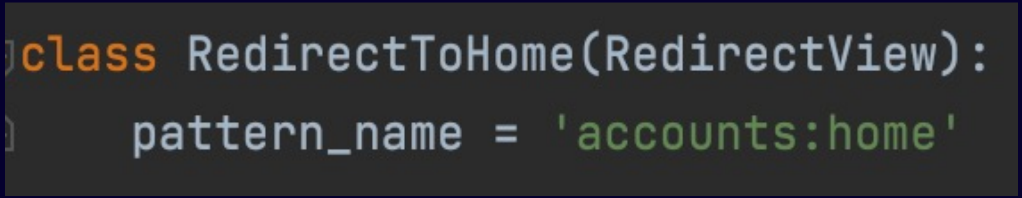
```python
def hello2(request):
    return TemplateResponse(request, "testapp/index.html",
                            context={'error': 'form is invalid'})
```



```python
def redirect(request, *args, **kwargs):
    return HttpResponseRedirect(reverse("accounts:home"))
```



```python
class SimpleView(TemplateView):
    template_name = 'testapp/index.html'

    def get_context_data(self, **kwargs):
        return super().get_context_data(**kwargs) \
               | {'error': 'form is invalid'}
```



```python
class RedirectToHome(RedirectView):
    pattern_name = 'accounts:home'
```

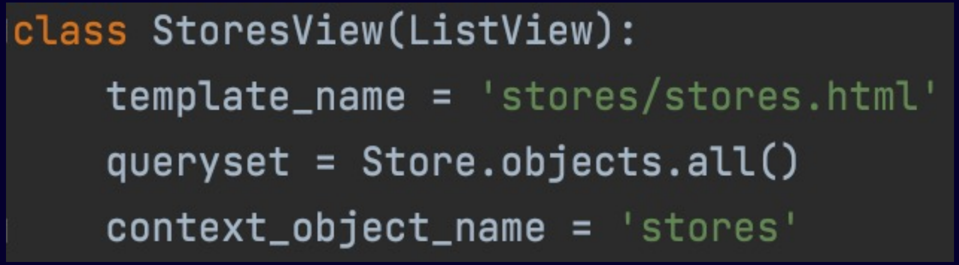
CRUD Views

- Stands for Create-Read-Update-Delete
- Many views fall under one of those categories
- Django’s CRUD views

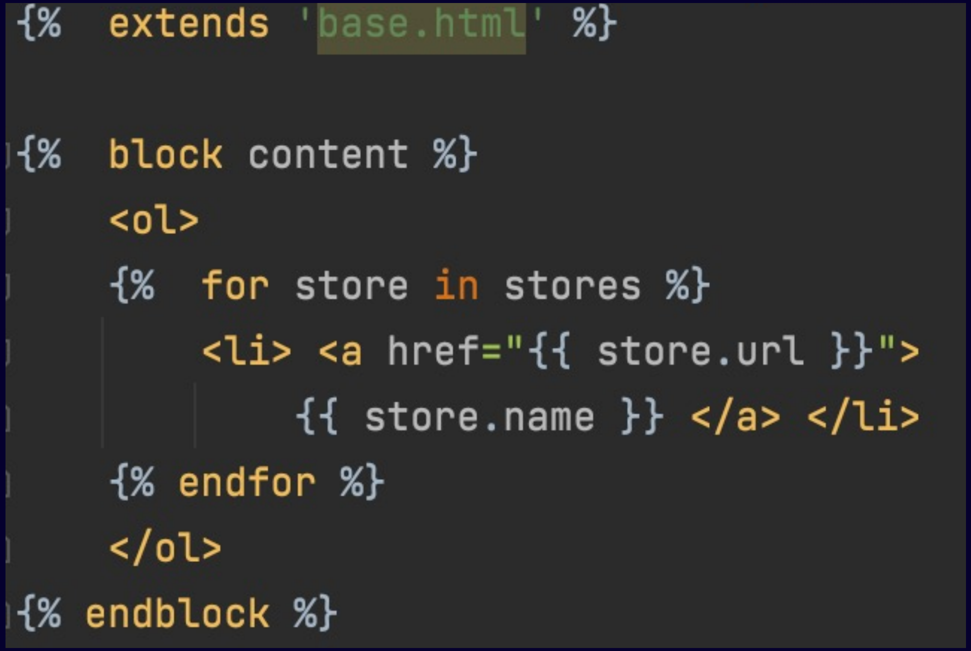
```
CreateView
DetailView, ListView
UpdateView
DeleteView
```

ListView

- You can specify a model or queryset attribute
- Resulting objects passed to the template context
- For more sophisticated cases, override get_queryset()



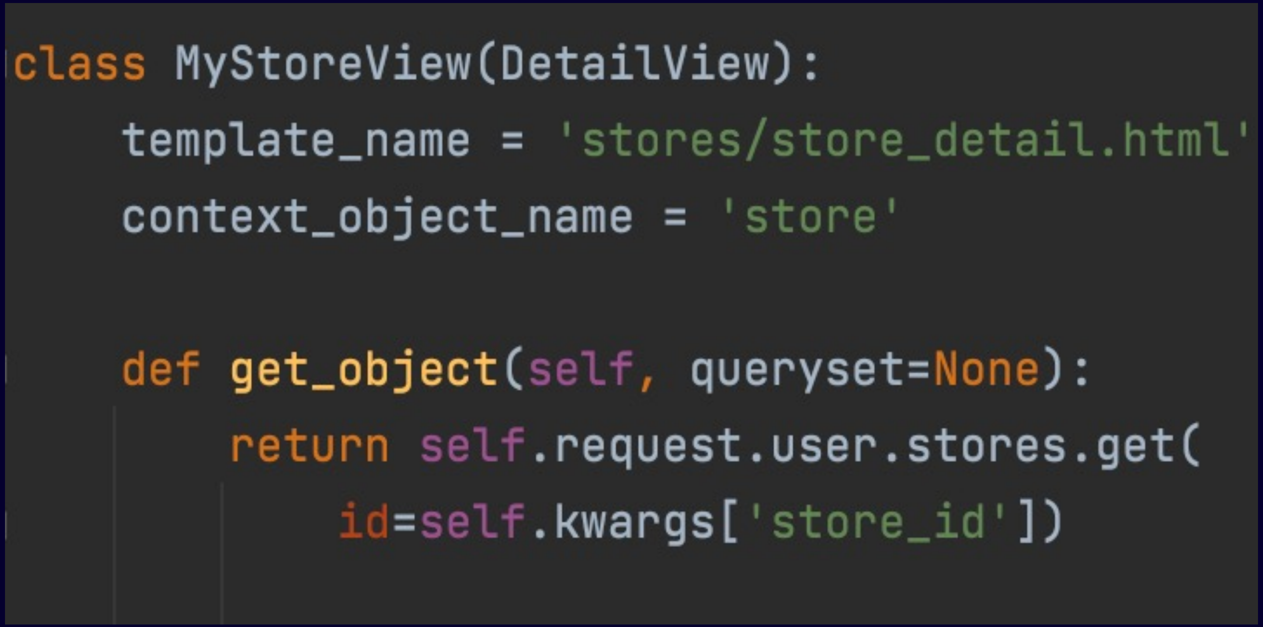
```python
class StoresView(ListView):
    template_name = 'stores/stores.html'
    queryset = Store.objects.all()
    context_object_name = 'stores'
```



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

{%  block content %}
    <ol>
    {%  for store in stores %}
        <li> <a href="{{ store.url }}">
            {{ store.name }} </a> </li>
    {% endfor %}
    </ol>
{% endblock %}
```

DetailView and DeleteView

- Shows the details of a single object
- Override `get_object()`
- DeleteView is similar to DetailView but implements the HTTP DELETE method



```python
class MyStoreView(DetailView):
    template_name = 'stores/store_detail.html'
    context_object_name = 'store'

    def get_object(self, queryset=None):
        return self.request.user.stores.get(
            id=self.kwargs['store_id'])
```

This session

- Custom models
  Fields, relations, queries
- Migrations
- Advanced views
  Class-based and CRUD views

Next session

- More CRUD
- Rest framework, JSON, and APIs
  - Serializers, permissions
- Django conclusion

Final notes

- Phase 1 deadline extended until Sunday
- Watch Piazza for interview sign-up
- Finish the whole Django tutorial
  https://docs.djangoproject.com/en/4.0/intro/tutorial01/