

DJANGO

Setting Up:

- `pip3 install pipenv`
- `pipenv install django`
- `pipenv shell` - python shell - use to start environment
- `django-admin`
- `django-admin startproject <project name> .` <- from the project directory
 - Init - defines the directory as a package
 - Settings - defines the applications settings
 - Urls - defines URLs of application
 - Other two modules - used for deployment
 - `manage.py` - wrapper around django admin and takes the settings into account
 - Run `python manage.py` - this will give a list of all commands available
 - Run `python manage.py runserver` - optionally takes a port number, by default 8000
- To get path to virtual environment: `pipenv --venv`
 - To tell VSCode where python is installed for the project, go to palette commands, search python interpreter, select enter path, and then past the path given by the command above + `/bin/python`
 - This makes it possible to use the integrated server in VSCode

• **VSCode Shortcuts:**

- Toggle terminal - `ctl + `` (backtick)
- Toggle left side panel - `cmd + b`
- Clear terminal window - `ctl + l`
- Go to a symbol, etc - `cmd + t`

- To start/create an app with the project – `python manage.py startapp <appname>`
`pipenv shell` <- first
- Remember to always add a new app to the `INSTALLED_APPS` list in the `settings.py` of the project
- Folders:
 - Migrations - generates database tables for project
 - Init -
 - Admin - define the admin interface
 - Apps - app configuration
 - Models - define model classes for app, to pull out data from database
 - Tests - for writing unit tests
 - Views - request handler
- Whenever a new app is created, it must be added to the list of installed apps in the settings module.

- **Views:**

- View function - takes a request and returns a response, a request handler, aka action
 - Views functions are mapped to URLs so that when a URL is accessed, that view function is called.
 - They are accessed `http://www.url.com/appname/functionname`

- **MAPPING URLs TO VIEWS**

- Map in application folder in a new file, commonly called `urls.py`
- Also need the following for path and accessing views:


```
from django.urls import path
from . import views
```
- Define `urlpatterns`, which consist of the route, which is a string, and the view, which is an HttpResponse object. This returns a URL pattern object. This is a URLConf, or configuration.
 - Always end these routes with a forward slash, (ex. urls.py of the app “playground”)


```
urlpatterns = [
    path('hello/', views.hello)
]
```
 - In the urls.py for the entire project:
 1. Import the `include()` function: `from django.urls import include, path`
 2. Add a URL to `urlpatterns`: `path('blog/', include('blog.urls'))`

urls.py for the entire project, app name playground:

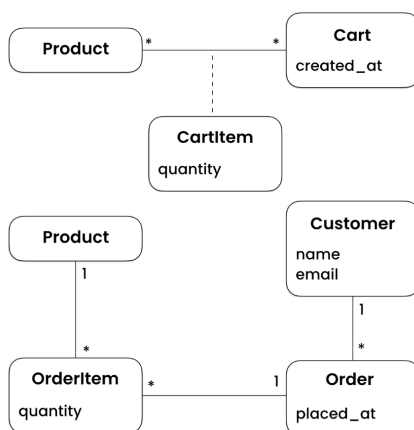
```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('playground/', include('playground.urls'))
]
```

- **TEMPLATES**

- Used to return HTML content to the client

- **Django Debug Toolbar**

- [Follow these steps](#)
- This will show up on any “proper” html page or template, so as long as it has an html and body tag around the content



MODELS:

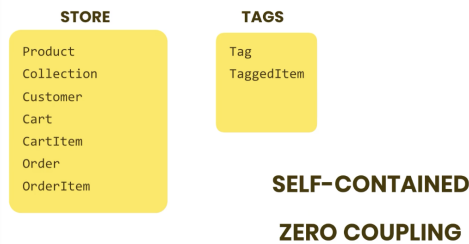
- Decide on the entities in the models as well as their relationships
- The relationship between cart and product is a many to many relationship, and both have a relationship to CartItem

Connecting Models to Apps:

- One app for each overall concept, self-contained for simple reuse
- Highly related concepts should be bundled together to decrease coupling between apps
- Multi-use apps that are not specific to a project should be separated out and not bundled.

If the app boundary is too large, it becomes a monolith that gets bloated and is hard to reuse. If the app boundary is too small, there is too much coupling between app.

A good design is one with minimal coupling and high cohesion, or focus, so that each app is focused on a specific functionality and includes everything needed to fulfill that piece of functionality.



- Self-contained: each provides a specific piece of functionality, so either or both can be used in another project, depending on what is being built
- Zero coupling: can independently change and deploy them without affecting other apps

Model Classes:

- Django Field Types for model fields, such as BooleanField, CharField, etc
 - Field types have all sorts of options that are available to all field types
 - Some field types have extra options
 - Ex. CharField has the option MaxLength which is required.
 - `price = models.DecimalField(max_digits=6, decimal_places=2)`
 - Always use DecimalField for prices. It requires the max total digits and decimal places
- Unique Id: For each class, Django automatically creates unique ID numbers. If you want to create your own, use `primary_key=True` within the creation of the field within the model

Choice Fields:

- Used for limiting the values that can be stored in a field.
- Use all uppercase to name the options list of tuples that are used as options for the choice field to indicate that it is a fixed list of values that should not be changed
 - Within the tuple, the first item is the field that the program will use to reference the choice, and the second is the human-readable version
- Within the choice field, set choices equal to the list of options.
- Choice fields also take a default option

One-to-One Relationships:

- Parent class should be stored before child class so that it can be referenced within the child
 - If it is not possible to define the parent first, then pass the parent class as a stringified version of the parent class name.
- Within the child entity, specify a parent
- Set the parent equal to `models.OneToOneField`
 - The required arguments contain the parent model and what to do on deletion
 - Ex: `models.CASCADE` will cause deletion of the child if the associated parent has been deleted, since this is a one-to-one relationship. Setting it to `models.SET_NULL` will leave the child in the database even if the parent has been deleted. It also can be `SET_DEFAULT` and given a default value, as well as `PROTECT`, which prevents deletion of a parent if there is a child present, requiring the deletion of any children first.
 - Setting `primary_key = True` on the parent field will avoid Django setting a unique id field and rather using the parent as the unique id.
- Between children and parent models, the relationship only needs to be defined within the child model. The reverse relationship is automatically contained within Django.

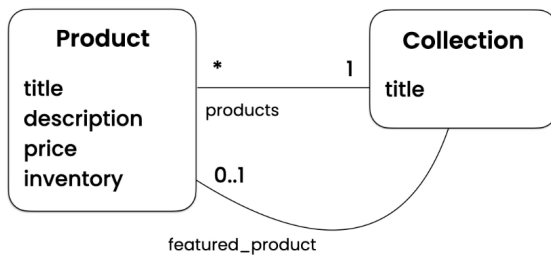
One-to-Many Relationships:

- Set field type to `models.ForeignKey` for the parent field within the child class

Many-to-Many Relationships:

- In the field that is most likely needed to bring in all of the related field items, use:
`variable = models.ManyToManyField(RelatedClass)`

Circular Relationships:



CIRCULAR DEPENDENCY

the class that explicitly states the relationship, the `related_name` can either be set equal to some other name to avoid the conflict, or set equal to `"+"`, which tells Django not to create the reverse relationship.

- Happens when two classes depend on each other at the same time. **This should be avoided.**
- Will usually create tricky situations in code where a class is defined after it is called by its dependent other class. This is when the first dependent class will need the related class **passed as a string** rather than variable name. This technique should only be used when completely necessary, due to the challenges of maintaining.
- When Django runs into a clash automatically creating reverse relationships (ex. a variable name has already been defined, so it cannot use it to define the reverse), within

Generic Relationships:

- When you have two separate apps with their own defined models that need to relate to one another somehow, which would require the import of one set of models into the other
- Instead, using generic relationships makes this cleaner, by defining a generic `content_type`, `object_id`, and `content_object` within one of the models to connect to the other.
 - These specific variable names must be used.
- This way, any record in a database can be located. Type gets you to the table, and id gets you to the record.
- The generic model for the `ForeignKey` is `ContentType` - this is installed as an installed app in Django, `contenttypes`
- By using a `GenericForeignKey` as well, we can read the actual object in the related table that the current object in our current table is related to.

DATABASES

- SQLite is good for development and low traffic sites
- PostgreSQL and MySQL are good for large project deployment

Creating Migrations:

- Migrations are used to create or update database tables based on the models we have created.
`python manage.py makemigrations`
- The files created here are translated into SQL from Python by Django

- Everytime models are changed, added to, or refactored, migrations must be run to update the databases built from them
- If you change the name of a migration file, be sure to change all references to that migration file in other migration files as well.
- If when migrating, Django asks for a default value and is given it in the terminal, it will not show up in the model. However this default can be set for good within the model and then migrated.
- Each migration describes a set of changes, like a commit in a version control system.
- Running the migrations is what actually generates the schema
- When creating migrations, it is best not to mix up operations and run a migration for each unique operation. Otherwise, the names Django gives the migrations are fairly vague and useless.

Running Migrations:

- To run all created migrations: `python manage.py migrate`
- The database is actually created as `db.sqlite3` in our directory
- This can all be viewed through SQLite in VSCode
- To see the sql code for a migration that has been run:

```
python manage.py sqlmigrate <appName> <migrationSequenceNumber>
```

Customizing Database Schema:

- We can add subclasses to a class in models, for example [Metadata](#), within a class

Undoing a Migration:

- To change individual effects of a previous migration, you must go and make the changes and migrate again.
- To undo all the effects of an entire migration:
 - `python manage.py migrate <appName> <migrationNumber>`
 - Also delete the migration file
 - Make all the changes back to the way it should be.
 - **OR** revert to previous commit before migration was applied.
 - See gits made so far: `git log -oneline`
 - Repoint head to previous commit: `git reset -hard HEAD~1`