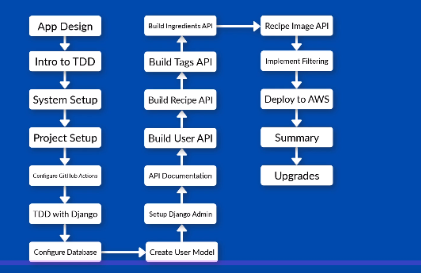
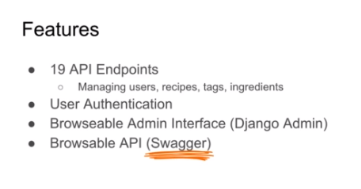
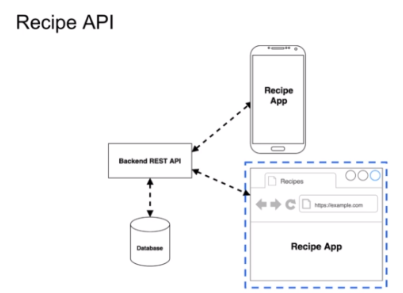
**Django API**

**Course Structure**

****

**Our Project**

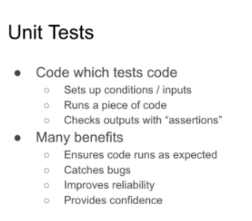
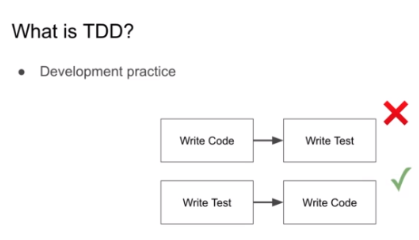
****

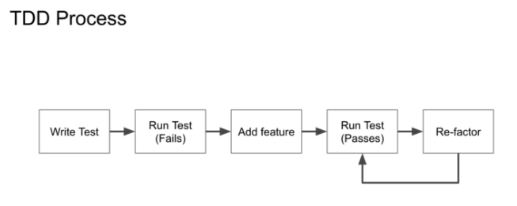
We use **Docker** a containerization software which allow us to run different application services

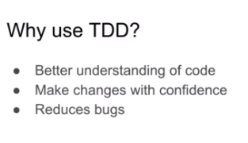
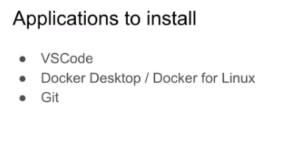
**Swagger API** generate automated documentation for our api

**Our project structure**

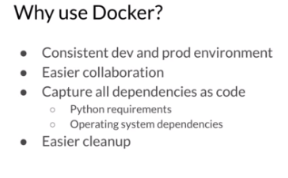
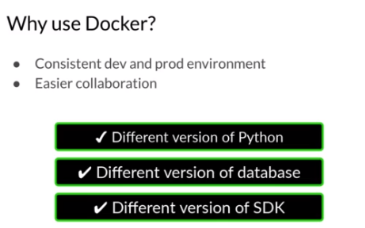


**What is Test Driven Development? **

****

**** ****

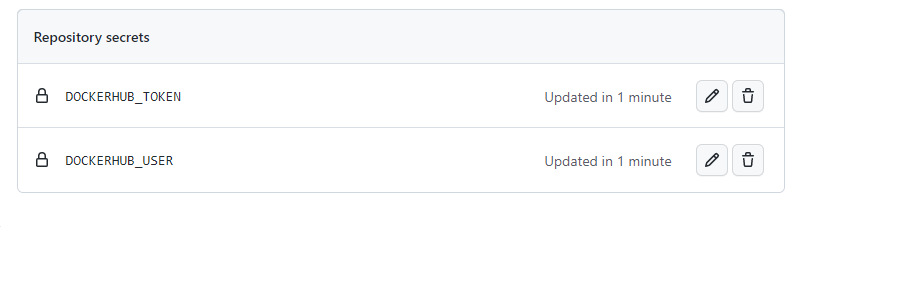
**** ****

****

**** ****

**1st we will create docker access token in your docker account and take token for later use in github secret and**

**2nd goto github 🡪 settings and 🡪 secret and create two things**

****

**In docker user section give your docker-user name and for token section give the token you get from docker account generated.**

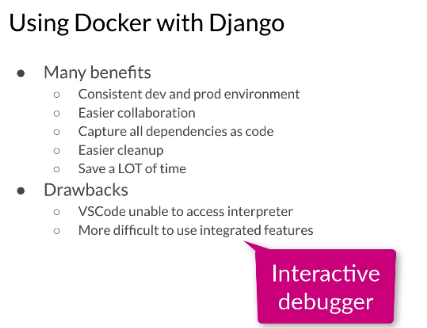
* **This two things is used in github action later in this course**

**In Docker Hub We will create a acces token to our project**

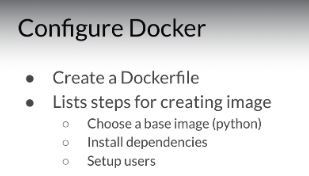
**And we will create secret key in our github account for our project**

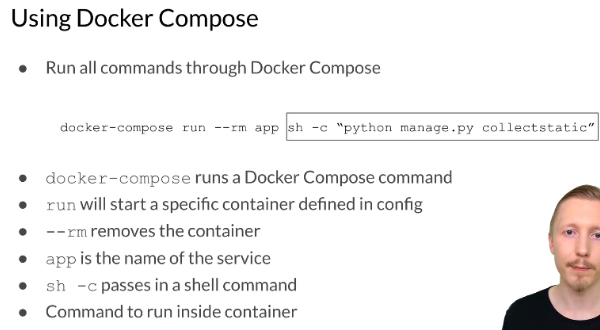
**In Docker Hub We will create a acces token to our project**

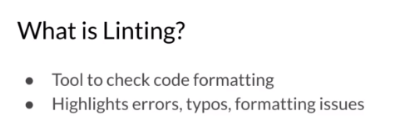
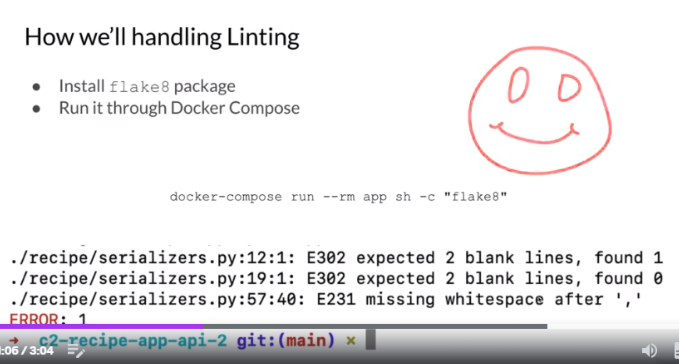
**And we will create secret key in our github account for our project**

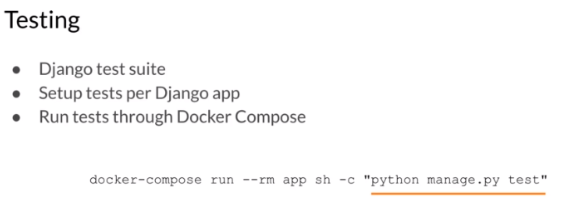
****

**Configuring Docker**

****

****

* **We have create requirement file and dockerfile for our project**
* **Docker composer consist of one or more services on it**
* **\recipe-app-api>docker-compose build**
* **\recipe-app-api>docker build .**
* 
* 

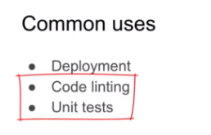
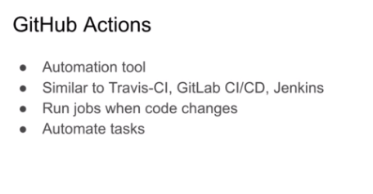


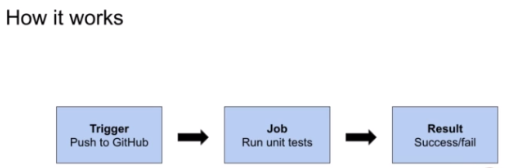
* **to run linting –**
* **docker-compose run --rm app sh -c "flake8"**
* **in order to create django project in app folder docker-compose run --rm app sh -c "django-admin startproject app ."**
* **. means in the current directory and after executing this command a new app folder will be created in app folder**
* **now we will run our django project using**
* **docker-compose up – will start all docker services**

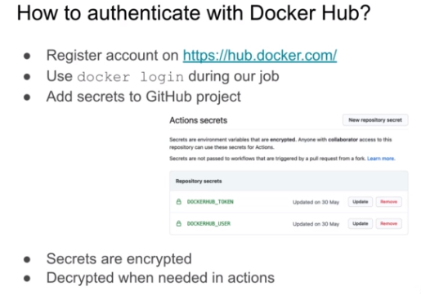
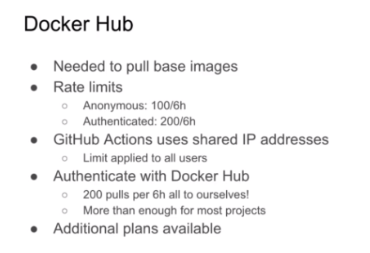
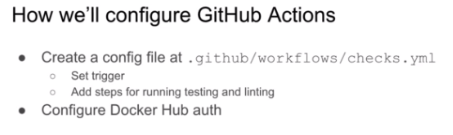
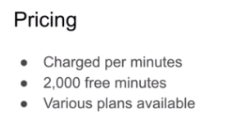
**Summary in this section**

* **we start by creating new project in github and in this setting we put docker hub secret key**
* **then we create project in our machine and create docker file docker-compose file and create requirement file for both deployement and development**
* **we are also create linting tool in app folder**
* **and create new blank django project in app folder**
* **and use docker-compose up to start our django application**

**Github Action**







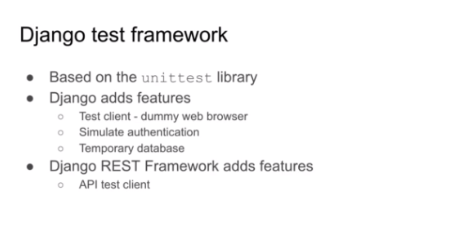
* **we start with \_\_\_ three dashes to start yml file**
* **this command used to run unit test**

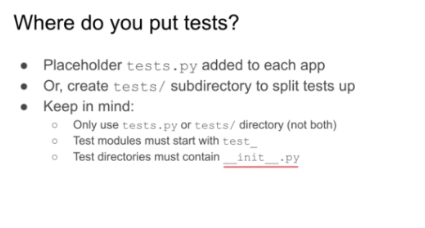
**docker-compose run --rm app sh -c "python manage.py test"**

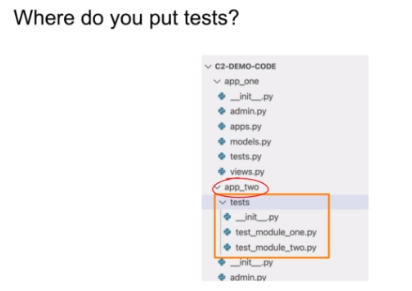
* **the below code also used to test lintng**
* run: docker-compose run --rm app sh -c "flake8"

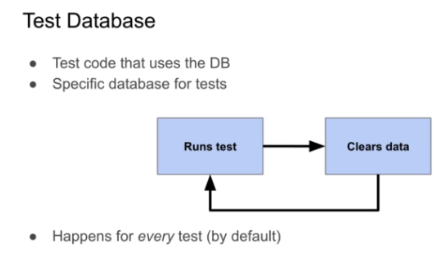
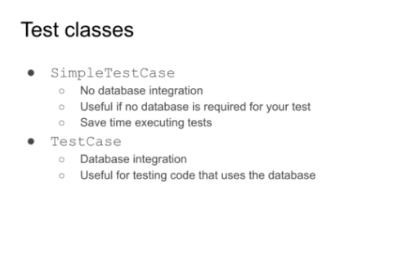
**in github/workflow/check.yml file we write a job that perform which push is performed in the github**

**Test driven Development**





**Sample example** 

* **We don’t need to create a test DB in our production db we will get a temporary test db by default**
* 
* 
* **Sample test code**

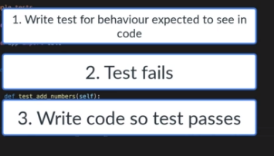
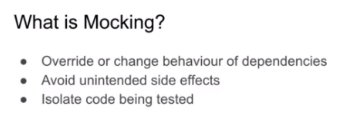


* **We will run the test command using docker command**

**docker-compose run –rm app sh -c “python manage.py test”**

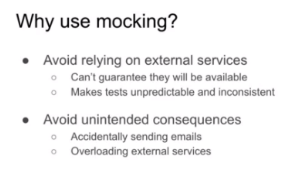
**run 🡪 indicate that we want to run service –rm indicate that to remove the container after finish running 🡪 and calling app service 🡪 sh -c is a shell command and run python manage.py test command**

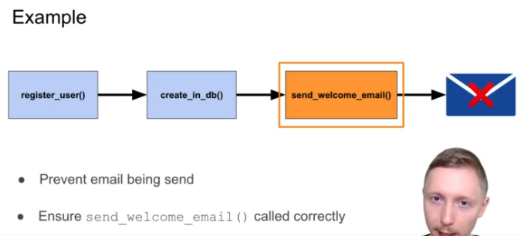
* **docker-compose run --rm app sh -c "python manage.py test"**
* **assertEqual() in Python is a unittest library function that is used in unit testing to check the equality of two values.**

**** 

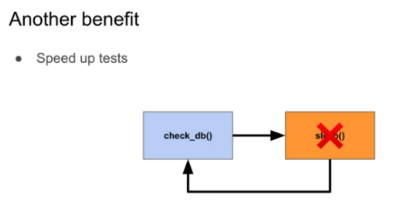
**Mocking is primarily used in unit testing. An object under test may have dependencies on other (complex) objects. To isolate the behaviour of the object you want to test you replace the other objects by mocks that simulate the behaviour of the real objects. This is useful if the real objects are impractical to incorporate into the unit test.**

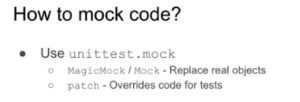
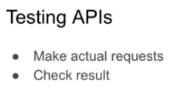
**In short, mocking is creating objects that simulate the behaviour of real objects.**

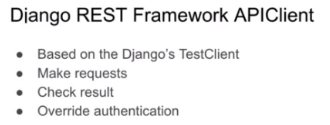
* **This is (mocking) is done for the purpose of unit testing**
* ****
* **Best Example for how to use Mocking in testing**

****

* **In the above example when try to test user registration functionality we have to mock send\_welcome\_email function and we know that the method is called correctly**

****

* **Let say we have a code that checks the db availability and it sleep and check again in certain period of time**
* **** ****
* **->using unittest.mock library**

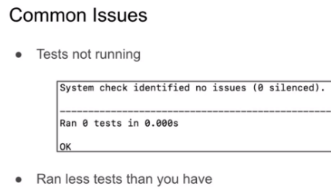
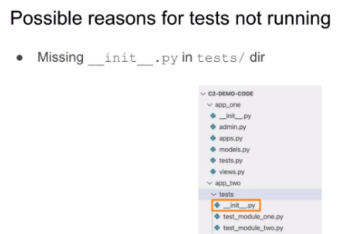
****

**Testing Web Request in Django sample code**

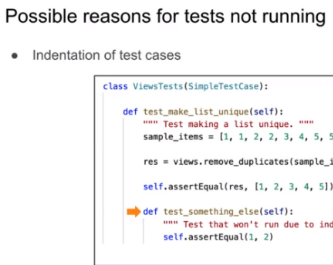
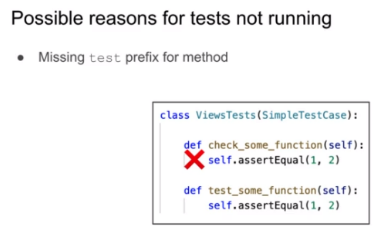
****

**Common problem happens in testing**

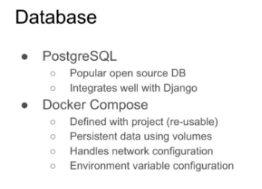
* **Sometimes we have 10 testing case but it runs 5 of them this is happens because of**

**** ****

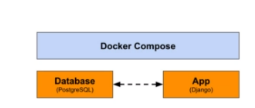
* The test runner always looking for test prfix method
* ImportError --- when running tests

**** ****

* **ImportError is happens because of having tests folder and test.py file in one same app**

**** ****

**How docker compose manage the service and its architecture**

**** ****

**In the above case the db service must be started first before starting app service**

****

**image: postgres:13-alpine 🡪 is official docker hub image file for postgres db**

**environment:**

**- POSTGRES\_DB=devdb**

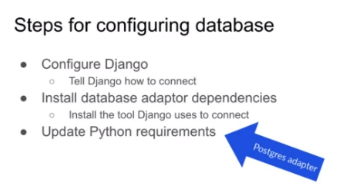
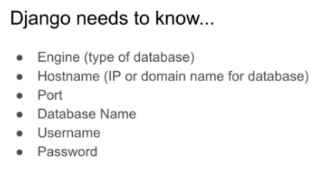
**- POSTGRES\_USER=devuser**

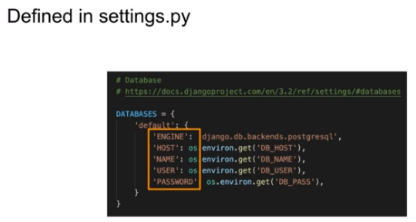
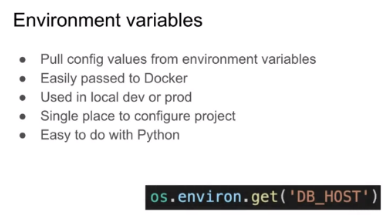
**- POSTGRES\_PASSWORD=changeme**

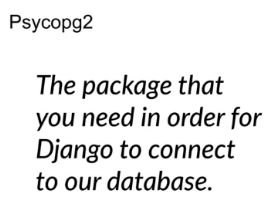
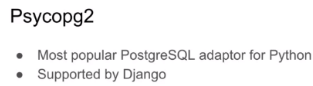
**🡪 this is environment variable when the db service starts for the first time it will create devdb database and creates db user named devuser with password changeme**

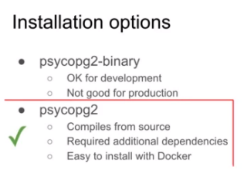
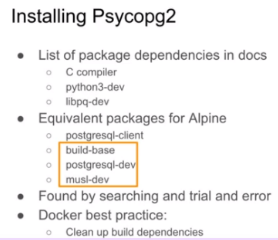
* **After setting up the configuration of db and app service environment variable we will run this command in terminal**

**Docker-compose up**

**** ****

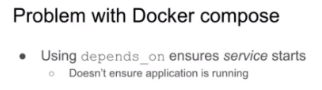
**** ****

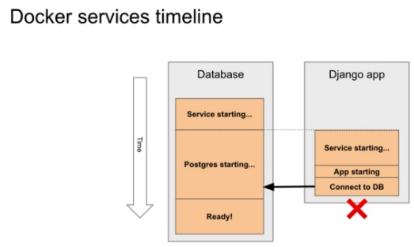
**** ****

**** ****

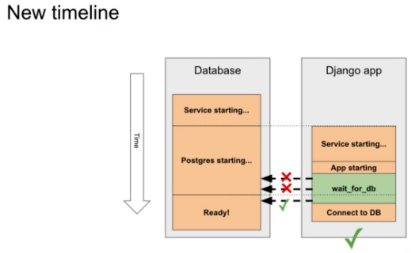
* **This command used to rebuild docker container and it will remove existing container configuration. docker-compose down**

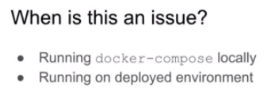
**Fixing Database Race Condition**

****

****

* **The above image shows that the when the database start it will take time while Django app start to connect with db and when the db is not ready the will crash for this problem the solution is wait command**

**** ****

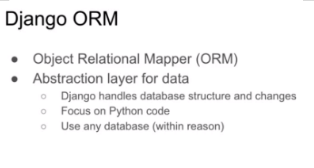
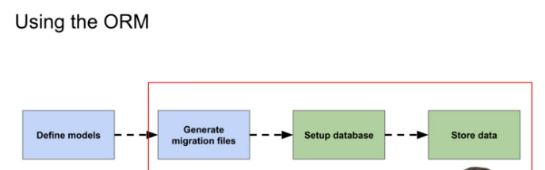
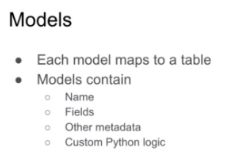
****

* **docker-compose run --rm app sh -c "python manage.py startapp core" ---- used to create new app in Django app**

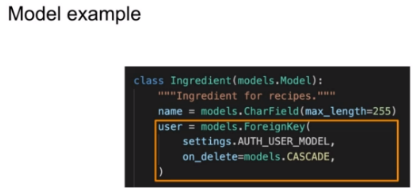
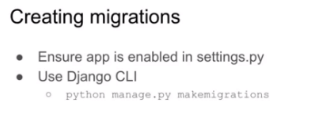
from django.contrib import admin  # noqa

**we will use this code when we are not using admin but we are import it then we use # Noqa to be ignored by flake8**

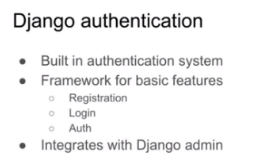
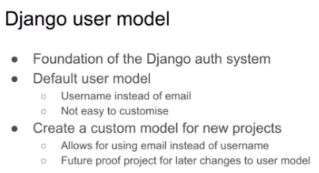
**Database Migration**

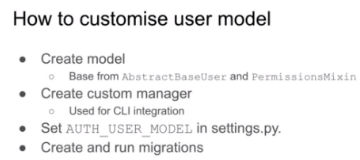
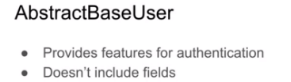
 

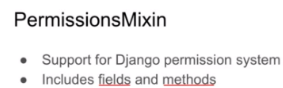
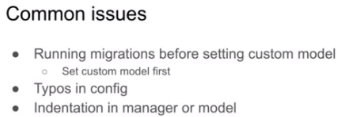
* docker-compose run --rm app sh -c **"python manage.py wait\_for\_db && flake8"**

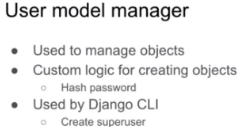


* we need to update docker-compose configuration for that the command for wait for DB runs before we start our server
* Handling Race condition – which means application wait until DB starts to avoid application crash
*  

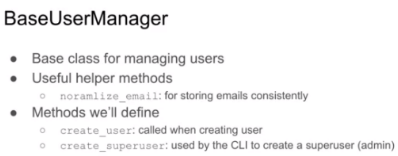
 

**The Design of Custom User Model**

**** ****

**Is staff -- attribute used to allow or deny user to login in Django admin**

****

* **When we are creating custom user model first we will create UNIT TEST Code**

**When we creating custom user model we will write this code in setting.py file**

* In order to create migration -- **docker-compose run --rm app sh -c "python manage.py makemigrations"**
* When we are migrating our migration to the DB use this command **docker-compose run --rm app sh -c "python manage.py wait\_for\_db && python manage.py migrate” which means first check first the DB is starting and migrate**
* To list volumes in our system using docker – **docker volume ls**
* And to remove data from volume in docker use this code – **docker volume rm filename but first we have to down docker either it will generate error file inuse**
* After migrating the model successfully and try unit test code

**Normalize Email Address**

* Adding feature to email in order to normalize

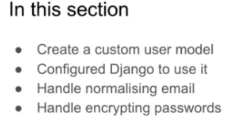
|  |
| --- |
|  |
| user = self.model(email=self.normalize\_email(email), \*\*extra\_fields) |
| Email input required also can be checked in Unit Test |  |  |

**Adding Superuser Functionality**

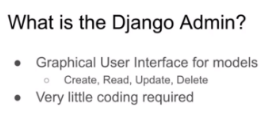
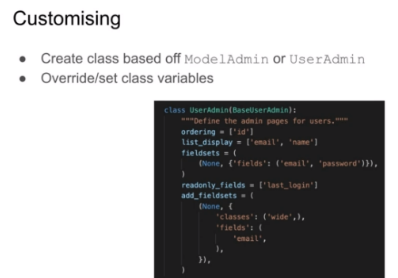
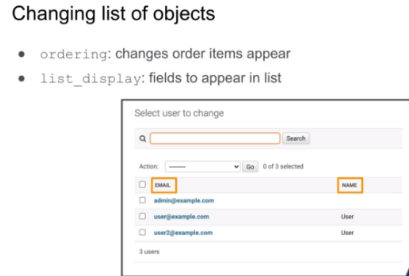
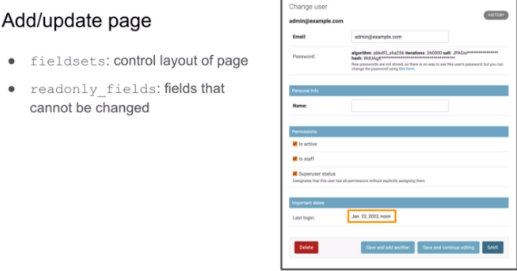
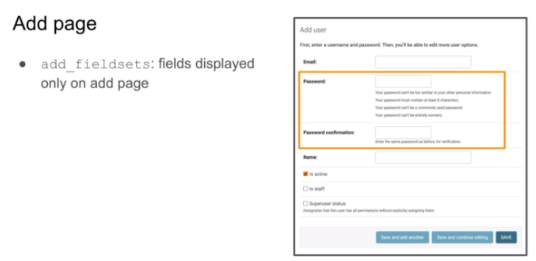
* **in order to access Django admin we need to have Is staff and Is super is must be true**

**Test User Model --- first we have create user using CMD using docker-compose run --rm app sh -c "python manage.py createsuperuser”**

**Summary**

* ****

**Setup Django Admin**

We have create Unit Test for Django Admin

**Writing Test for listing user in admin**

**Admin\_test.py file has unit test for admin site**

## Reversing admin URLs[¶](https://docs.djangoproject.com/en/3.1/ref/contrib/admin/#reversing-admin-urls) read the following link

[**https://docs.djangoproject.com/en/3.1/ref/contrib/admin/#reversing-admin-urls**](https://docs.djangoproject.com/en/3.1/ref/contrib/admin/#reversing-admin-urls)

from django.utils.translation import gettext\_lazy as \_

**the above code used in Django to translate to other language**

class UserAdmin(BaseUserAdmin):

    """Define the admin pages for users."""

    ordering = ['id']

    list\_display = ['email', 'name']

    fieldsets = (

        (None, {'fields': ('email', 'password')}),

        (\_('Personal Info'), {'fields': ('name',)}),

        (

            \_('Permissions'),

            {

                'fields': (

                    'is\_active',

                    'is\_staff',

                    'is\_superuser',

                )

            }

        ),

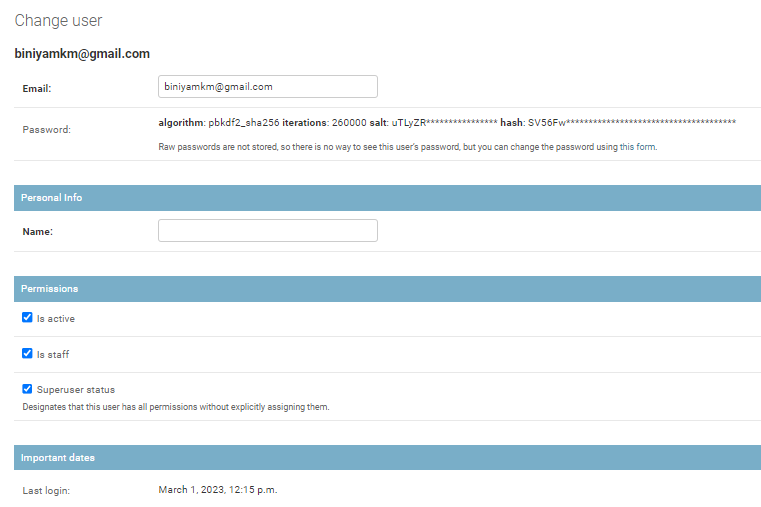
        (\_('Important dates'), {'fields': ('last\_login',)}),

    )

    readonly\_fields = ['last\_login']

admin.site.register(models.User, UserAdmin)

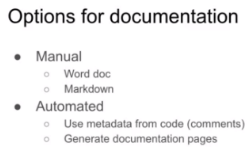
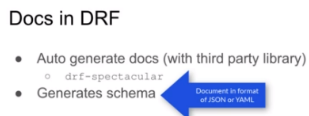
**the above code enable us to see and update user information like below**

****

* ***Test on Development is the best thing for error tracing***

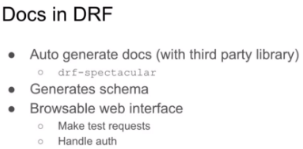
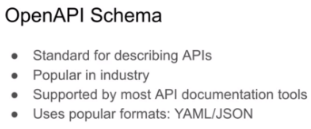
**API Documentation**

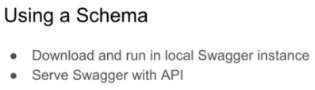
 

**How it works**

1. **Generate “schema” file**
2. **Parse schema into GUI**

**Install DRF- Spectacular**



To Enable DRF-Spectacular in url use this code

from drf\_spectacular.views import (

    SpectacularAPIView,

    SpectacularSwaggerView,

)

path('api/schema/', SpectacularAPIView.as\_view(), name='api-schema'),

    path(

        'api/docs/',

        SpectacularSwaggerView.as\_view(url\_name='api-schema'),

        name='api-docs',

    )

