General notes

- Overall I think this looks pretty solid, keep the up the good work!
- Nice naming of models, you have also thought about common fields like created_at and updated_at (which is absolutely not a given in my experience...)
- Nice work lately improving error messages (for example), this type of work is often overlooked, but very helpful both for end users and for people trying to understand what happened when something went wrong.

Gitlab usage - branching strategy

I suggest that you try to make use of the main branch + feature branches, instead of the current setup with a dev branch and "sleeping" (not very active) main branch. In order to make this work, you should also make use of "gitlab environments", where you can deploy branches more freely. So instead of having a dev, int and prd branch, you can have 3 environments and then chose which one to deploy a branch to (typically you are only allowed to deploy the main branch to the prd environment).

The dev environment could for example use the sqllite database, to allow for easy testing, while int should have a "real database" where any model migration scripts can be tested, so that they are production ready.

Gitlab environments are configured in the .gitlab-ci.yaml file, I'll provide an example on how it can look further into this segment.

Why?

Working like this will allow you to: - Work in a more iterative way, in a larger scale since this way of working encourages smaller changes that are tested immediately and then shipped to prd. - Paves the way for code reviews in the future (or now) - Merging into the master branch will provide some documentation of what's been going on with the project, with very little overhead. Instead of a commit history filled with "update", the main branch will have commits clearly connected to a work item, for example "merge of feature/add-course-model". (In the feature branch, there can be as many "update" commits that you want, I tend to squash them on merge, but that is a contested topic in the industry...)

Even if you chose not to follow the main + feature branch strategy, I strongly suggest that you start to make merge requests towards the dev branch, for the same reasons outlined in this segment

How

1. Keep all production ready code in the main branch.
2. Whenever it is time to add a change, or fix a bug, create a feature branch from the main branch. This could for example be called feature/add-course-model, or hotfix/fix-course-model-bug. If you have an issue tracking system, try to reference the ticket name or ID in the branch name, like feature/SX-XXXX-add-course-model, with jira it is for example possible to link commits into a jira ticket.
3. Create a merge request towards main.
4. Review the code, hopefully together with a peer, also test the code in an appropriate environment, if it is a complex change, it can sometimes help to provide "proof" that the MR is doing what it is supposed to do, for example a screenshot or test report.
5. If everything is fine, merge to main, and try to deploy it is as soon as possible, deploying small changes often, will reduce risk of a deploy and make it easier to troubleshoot problems.

Gitlab environment example

Gitlab environments here and Another good article here .gitlab-ci.yml (Please note that there might be syntax issues in this example :smile:)

```
.dev:
  variables:
    - ENV_SPECIFIC_VARIABLE_1
  environment:
    name: dev

.deploy-job:
  stage: deploy
  tags:
    - "Eddi-backend"
  variables:
    #GIT_STRATEGY: clone
    BUILD_DIR: "/home/admin/n5qVYxG8/0/e3673di/eddi-backend/"
    TARGET_DIR: "/var/www/html/eddi-backend"
  script:
    - echo "Deploying application..."
    - rsync -P -av $BUILD_DIR/* $TARGET_DIR
    - cd $TARGET_DIR
    - pip3.9 install -r requirements.txt
    - echo "Application successfully deployed."
    - rm -rf /home/admin/n5qVYxG8/
  only:
    - supplier-dev

deploy-job-dev:
  extends:
    - .dev
    - .deploy-job
```

Adding a new env from here is as simple to just add a new ".ENV" block, and then a new env specific step, like "deploy-job-ENV".

Django models and apps

The project is currently organized in one "Django App". It is a large project with many different features for different stakeholders - customers, suppliers, site admin + CMS. This results in a lot of models. While I don't really have anything to say about them, they have good names, both on model and field level, I think you can use some concepts to simplify a little, making future development easier. I think it is reasonable to make these changes in the order listed here.

1. Inheritance - There are a lot of common fields in many of the models, create an abstract base class with these fields instead of including them in every model, see the docs here: https://docs.djangoproject.com/en/3.2/topics/db/models/#abstract-base-classes, example below. Adding an abstract base model means that a table wont get created in the database, just the child models will get tables.

```
from django.db import models

class CommonInfo(models.Model):
  name = models.CharField(max_length=100)
  age = models.PositiveIntegerField()

  class Meta:
      abstract = True

class Student(CommonInfo):
  home_group = models.CharField(max_length=5)
```

This change should reduce the amount of rows a little in the models.py file.

2. Split the models into multiple files, this will make it easier to maintain these in the future (touching a 1500 lines file is always scary). https://docs.djangoproject.com/en/4.0/topics/db/models/#organizing-models-in-a-package
3. Multiple django apps - Identify parts of the backend to split into different django apps. My suggestion is to start with CMS + rest of backend, and then iteratively split the backend into more apps, for example depending on domain or end user (customer, supplier, site-admin). You can see an example of how it can look like here, imagine "posts" being "cms" and then that there is another, similar folder structure called "backend" or "supplier". Since you already split the models files, it should be pretty easy to move them into their own apps.

Supplier views

The supplier_views.py looks like a very complicated file.

- The functions have a lot different conditions and nested functionality, for example https://gitlab.com/e3673di/eddi-backend/-/blob/supplier-dev/eddi_app/supplier_views.py#L269. It should be possible to create private helper functions to hide some of the logic and make it easier to follow. And also test...
- Split the file and the classes into multiple files, maybe even one file per entity, and put them into a folder structure - views/supplier/get_course_details.py

Tests

Start adding unit tests. There is a lot of functionality, I strongly suggest that you should start locking it down using unit tests that are run when pushing code and deploying. This is some massive work, but very helpful in the long run. You don't need to do it all at the same time, take some time each sprint to do it.