CSC309H1S

Programming on the Web

Winter 2023

Lecture 6: Custom Models and CRUD

Instructor: Kuei (Jack) Sun

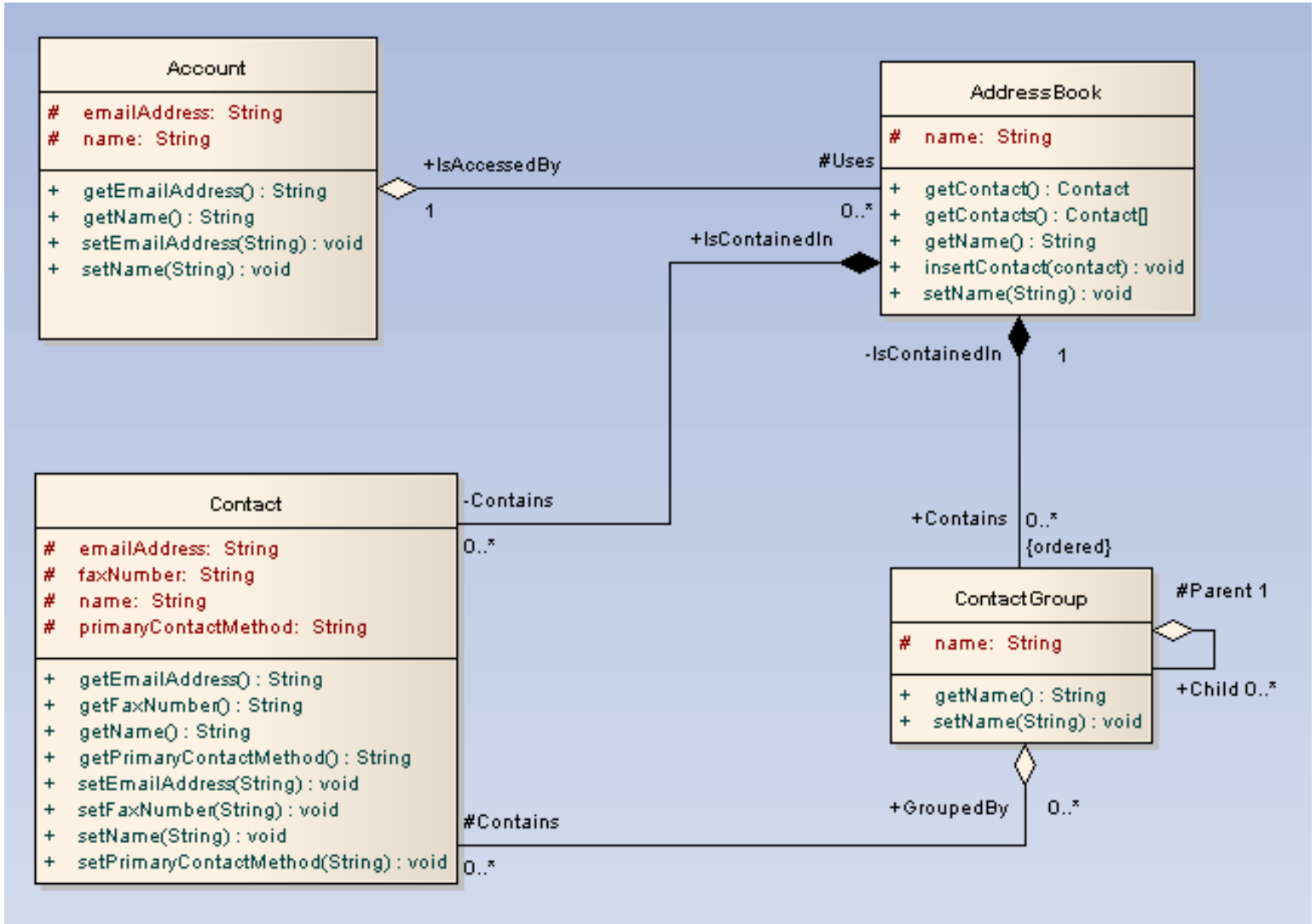
Department of Computer Science
University of Toronto

v1.02

Designing Models

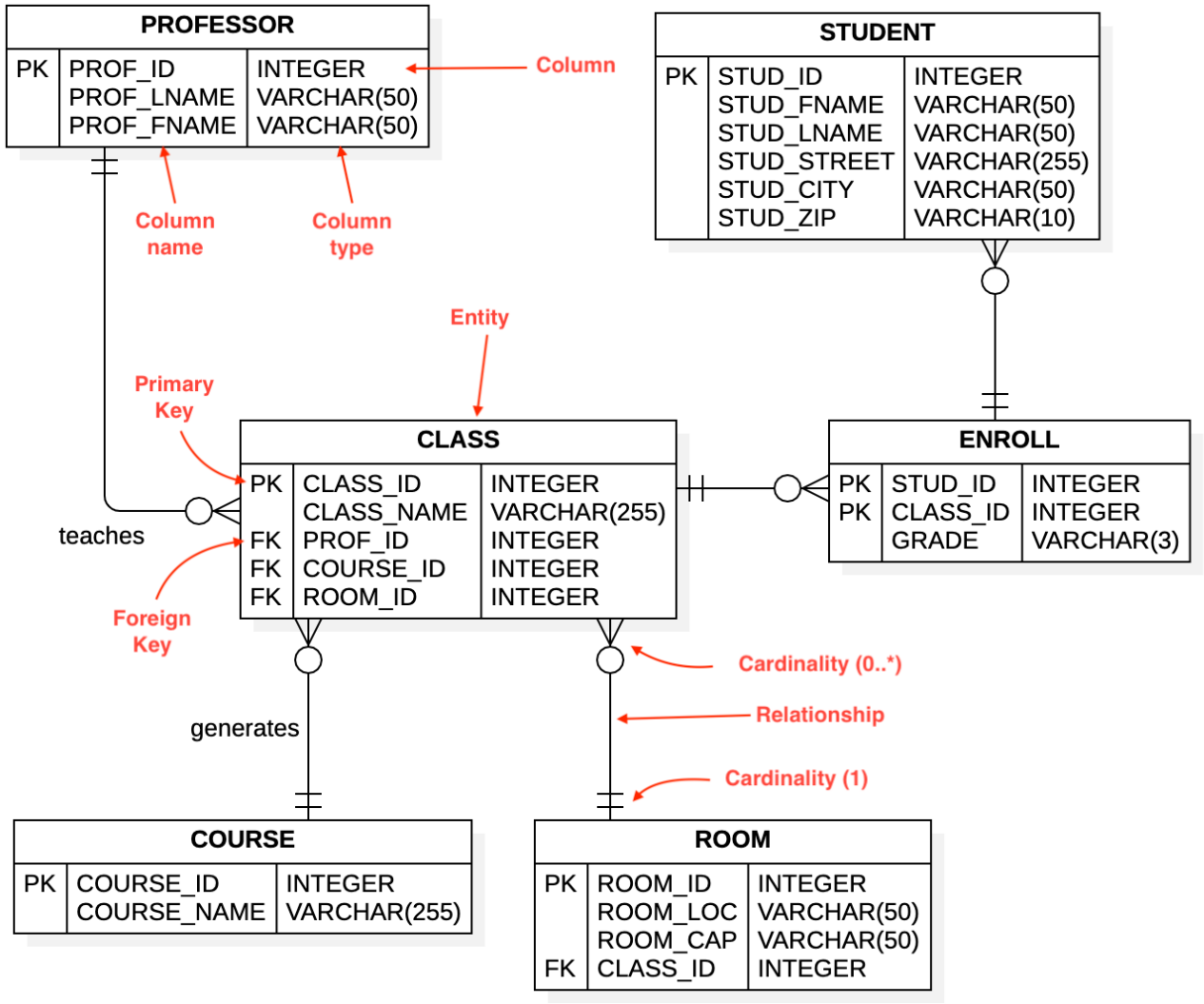
- Models involve representing and storing user data
  - A secure and efficient design is paramount to the success of an application
- General practice
  - Modeling should be done *before* coding starts
  - Changing models becomes more difficult when further into development cycle
    - Especially during the production phase
- Can be done independent of programming language or framework
  - E.g., Universal Modeling Language (UML)
    - Provides graphical notations to visual the design of an application
    - Not learned in this course, take CSC301
  - E.g., ER (entity relationship) diagrams

UML Diagram



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

Entity Relationship Diagram (ERD)



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

Django Models

Creating Models

- In models.py, add subclasses of `django.db.models.Model`
- Add fields from your diagram to each model
  - https://docs.djangoproject.com/en/4.1/ref/models/fields/
  - Each field is mapped to a database column by the ORM layer

```
from django.db import models
class Store(models.Model):
    name = models.CharField(max_length=255)
    email = models.EmailField()
```

- Convention for large projects:
  - Create a models directory and put each model in a separate file
    - Add __init__.py and *import each model*

Django Fields

- Each field type maps to a primitive type in the database
- Strings
  - CharField
    - For small amount of text
  - EmailField, URLField
    - Checks for valid format
  - TextField
    - For large amount of text
- Files
  - Need to specify where to save them
  - FileField, ImageField
- Numbers
  - IntegerField
  - BigIntegerField
    - For large numbers (at least 64-bit)
  - FloatField, DecimalField
    - Maps to Python float and Decimal
- Time
  - DateField, DateTimeField
- True/False
  - BooleanField

Django Field Options

- Every field can be restricted/checked in some ways
- `null: bool = False`
  - Used to allow the lack of value
- `blank: bool = False`
  - Allows field to be unspecified
  - Automatically given *empty* value
    - E.g., zero, empty string, etc.
- `unique: bool = False`
  - Requires value to be unique
    - Throughout the database table
  - Otherwise throws IntegrityError
- `choices: [(Any, Any),…]`
  - A list of key-value pairs
  - Key should be an abbreviation
  - Value is what is displayed to user
- `max_length: int`
  - Only for CharField or its subclasses
  - Limits the number of characters
- `default: Any`
  - Default value for the field

Foreign Key

- `models.ForeignKey(to, on_delete, related_name, …)`
- Used for many-to-one and one-to-many relations
  - Defined at the "many" end as part of the foreign key
  - Stores only the primary key in the database column
- `on_delete`
  - Determines the behavior when referenced object is deleted
    - CASCADE: delete everything that references the deleted object
    - SET_NULL: set reference to NULL. (Used to support 0..1 relationship)
- `related_name`
  - Provides alternative name for reverse traversal by a field
    - Default is <model_name>_set, e.g., `prof.class_set.all()`

Foreign Key

```
class Professor(models.Model):
    name = models.CharField(max_length=255)


class Class(models.Model):
    room = models.CharField(max_length=255)
    capacity = models.IntegerField()
    instructor = models.ForeignKey(Professor, on_delete=models.SET_NULL,
                                   null=True, related_name="classes")
    course = models.ForeignKey('Course', on_delete=models.CASCADE)


class Course(models.Model):
    code = models.CharField(max_length=16)
```

Does not delete the class when the professor is deleted. Requires null=True.

Forward reference must be specified in a string

deletes the class when the course is deleted

Models to Tables

- Every time the model changes, you must create and run migrations
  - `./manage.py makemigrations`
  - `./manage.py migrate`
  - More on this later
- By default, Django creates an AutoField named `id`
  - Configurable in <app_name>/apps.py
  - It is used as the primary key for the table
  - Can be overridden with `primary_key=True` for another column
    - Not a good idea in general
- Exercise 5
  - Do question 2 on Quercus

Admin Panel

- https://docs.djangoproject.com/en/4.1/ref/contrib/admin/
- Register your model to the admin panel
  - In admin.py add:
    - `admin.site.register(Store)`
- Field options for admin panel (and also form)
  - help_text: adds help text in tooltip
  - verbose_name: alternative name for the field
- `__str__()`
  - Called when typecasting Python object to str, i.e., str(obj)
  - Admin panel does this when displaying a list of objects

Model Inheritance

- OneToOneField
  - Defines a one-to-one relationship
  - Not frequently used, same thing can be done with *inheritance*
- Model inheritance
  - Creates an additional table with a pointer to the base class
  - Too many levels of inheritance can reduce performance

```
class Product(models.Model):
    name = models.CharField(max_length=255)
    price = models.FloatField(default=0.,)
    store = models.ForeignKey(Store, on_delete=models.CASCADE)

class Produce(Product):
    expiry_date = models.DateTimeField()
```

Many-to-Many Relationship

- ManyToManyField
  - Defines a many-to-many relationship, e.g., classes and students
  - By default, an intermediary join table is used to represent the relationship
  - Also supports recursive relationship (ForeignKey can do the same)

```
class Student(models.Model):
    friends = models.ManyToManyField("self")
```

Symmetrical relationship

- Can specify the intermediary table manually

```
class Student(models.Model):
    classes = models.ManyToManyField(Class, through='Enroll')

class Enroll(models.Model):
    student = models.ForeignKey(Student)
    klass   = models.ForeignKey(Class)
    grade   = models.CharField(max_length=3)
```

Working with Relationships

- add method
  - Associate two objects in a one-to-many or many-to-many relationship

```
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)
```

- Can access foreign object(s) through current object
  - May require multiple database queries, so be mindful of performance

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

  - `select_related(fieldname)`: grab related object in a single query

```
Store.objects.select_related('owner').get(id=1)
```

File Upload

- Only the file's local path is stored
- In settings.py, create the media root folder and its URL

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

  - By default, the upload_to folder is created in the project directory!
- Browser sends a separate request to access the file
  - Django translate request to a file access
- For images, must install the pillow package
- To access uploaded files, must register with URL dispatcher

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

Migrations

ORM Layer

- Assumption
  - The state of database tables is the same as the definitions in model classes
- Reality
  - The two can become out-of-sync whenever the model changes
    - i.e., database tables are independent of the model classes
- ORM must apply *current* application schema to the database
  - This requires running DDL (data definition language) queries
    - E.g., CREATE, ALTER, TRUNCATE, or DROP commands
    - E.g., create/remove a table
    - E.g., create/remove a column due to updated model
    - E.g., change column options due to updated field attributes

Migration

- Whenever model changes, database should migrate to the new state
- Django does not perform migration automatically.
  - Why?
    - Quercus Exercise 5 Question 7
- Django does not monitor potential model changes
  - If there's a mismatch, database exception will occur when executing queries
    - E.g., selecting a table/column that does not exist
- There are two steps to perform migration
  1. Make migration
  2. Migrate

Make Migration

- Generates a list of operations needed to migrate to new state
  - Similar to a git commit
    - It tracks what has changed since the last migration

```
class Migration(migrations.Migration):
    dependencies = [ ('products', '0001_initial’), ]
    operations = [ migrations.AlterField(
            model_name='product’, name='name',
            field=models.CharField(max_length=128),
    ),]
```

- The history of changes is stored in the migrations folder for each app
  - Each migration also tracks a list of *dependencies*
- To generate a migration, run: `./manage.py makemigrations`

Make Migration

- Builds a *temporary* model state from previous migrations
  - By replaying all migrations in order, e.g., from 0001 to current
- No data operations are executed
  - Migrations are written in a database-neutral way
- Temporary model state is compared with current model state
- From the differences, a list of operations is generated
  - A new migration file is created with a Migration class defined
    - E.g., 0003_delete_produce.py
- Note:
  - __init__.py must be present in the migrations folder for the command to work

Applying Migrations

- `./manage.py migrate`
  - DDL queries are generated from each migration file
  - Then, they are executed to apply the migration
- How does Django know which migration has not been applied?
- Migration information is stored in the database
  - In a table named django_migrations
  - Table only includes metadata, e.g., name, app, time applied, etc
    - Actual operation is stored in the migration files
- The migrate command only applies migrations not in the table

Migration Error

- You should *never* need to manipulate migration files/table
  - Let Django ORM do its things
- Lots of assumptions go into its implementation
  - E.g., deleting a migration file may permanently break future migrations
- Migrate errors can take many hours to resolve
  - Can occur when same model is modified by multiple developers
    - E.g., similar to resolving conflicts during git merge
  - Typically avoided because migrations have dependencies
- Possible solutions
  - You can *unapply* or *fake* a migration

Unapply Migration

- `./manage.py migrate <app> <last_migration_name>`
  - Rolls back all changes to a previous migration state
    - Data loss is possible. Back up is recommended
      - E.g., created tables may be deleted, new columns may be deleted
    - Create a full backup of the database in a JSON file
      - `./manage.py dumpdata > db.json`
    - Load the database with data from backup file
      - `./manage.py loaddata db.json`
    - More information: https://docs.djangoproject.com/en/4.1/ref/django-admin/
- The corresponding row in django_migrations is deleted
  - You may then delete the migration file that was unapplied
    - **Do not delete** a migration file before it is unapplied

Migration Tips

- `./manage.py migrate --fake`
  - Only creates a row in django_migrations
    - Without executing any database queries
  - Use case
    - When the database state is already consistent with the models
- Full reset
  - Last resort, if the migrations are becoming too messy
    - Remember to make a backup if there are needed user data in the database

  1. Delete the entire database
     - E.g., delete db.sqlite3
  2. Delete all migrations files to start over from fresh

Advanced Views

Class-Based Views

- Function-based views can become too big
  - One view may need to support multiple HTTP methods
- Class-based view
  - A subclass of `django.views.View`
  - HTTP requests are routed to methods of the respective names
    - E.g., HTTP GET request will call `view.get(request, …)`
  - A new instance of the object is created for every request
- Convention for large projects
  - Create a views directory and put each view in a separate file
    - Add __init__.py and *import each view*
    - In urls.py, each class-based view must call the `as_view()` method

Comparison Between Views

- Function-based views

```
def simple_view(request, id):
    if request.method == "GET":
        return HttpResponse(
                f"My ID is {id}")
    elif request.method == "POST":
        return redirect("accounts:login")
    else:
        return HttpResponseNotAllowed()
```

- Class-based views

```
from django.views import View
class SimpleView(View):
    def get(self, request, id):
        return HttpResponse(
                f"My ID is {id}")

    def post(self, request, *a, **k):
        return redirect("accounts:login")
```

- In urls.py

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

CRUD Views

- Create-Read-Update-Delete
- Most views fall under one of these categories
- Django provides CRUD base classes for these views
- Generic display views
  - Designed to display data
  - DetailView and ListView
- Generic editing views
  - https://docs.djangoproject.com/en/4.1/ref/class-based-views/generic-editing/
  - Designed to create, update, or delete data
  - CreateView, UpdateView, DeleteView, FormView (next week)

List View

- A page that displays a list of objects
- https://docs.djangoproject.com/en/4.1/ref/class-based-views/generic-display/

- View

```
from django.views.generic.list \
    import ListView
from .models import Store

class StoresList(ListView):
    model = Store
    context_object_name = 'stores’
    template_name = 'stores/list.html'
```

Chooses which template to render

- Template

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

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

Display View Attributes

- models
  - The model of the generic view
  - Assumes query set is entire table
- queryset
  - Same as model, but allows you to specify a subset or ordering

```
Store.objects.filter(is_active=True)
```

- get_queryset(self)
  - Override to customize query set
- URL arguments stored under `self.kwargs` e.g., `self.kwargs['pk']`
- context_object_name
  - By default, this is object_list or object (DetailView)
  - Allows alternative context name
- get_context_data(self, **kwargs)
  - Override to add extra context
- get_object(self)
  - DetailView only
  - Override to retrieve object
- Request object stored under `self.request`

Create View

- A page that allows for creating objects
- On GET request, returns blank form
- On POST request, redirect on success, redisplay form upon error

- View

```
from django.views.generic.edit \
    import CreateView
from .models import Store

class StoresCreate(CreateView):
    model = Store
    template_name = 'stores/create.html'
    fields = ['name', 'url', 'email', \
              'owner']
    success_url = \
        reverse_lazy('stores:list')
```

- Template

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

{% block content %}
<form method="POST">
  {% csrf_token %}
  {{ form.as_p }}
  <input type="submit" value="Create">
</form>
{% endblock %}
```

Editing View Attributes

- fields
  - A list of fields in the model to edit
  - Does *not* have to be every field
- success_url
  - Redirect URL when success
  - Must use `reverse_lazy` here
    - URL dispatcher loaded later
- get_success_url(self)
  - Needed if `reverse` needs argument

```
reverse('stores:detail',
    kwargs={'pk' : self.kwargs['pk']})
```

- Django form
  - Helps with all aspect of form
    - E.g., render HTML, validation, update associated model
- All edit views have a `form` context

```
{{ form.as_p }}
```

⬇

```
<p>
  <label for="id_name">Name:</label>
  <input type="text" name="name" maxlength="40"
         required id="id_name">
</p>
<p>
  <label for="id_url">Website:</label>
  <input type="url" name="url" maxlength="200"
         required id="id_url">
</p>
```