Módulo 7 - Aprendizaje Esperado 4: Ejercicio Individual

Nombre: Cristian Aranda Bórquez

1. Conceptos Fundamentales

Explica en tus propias palabras qué es una migración en Django.

Una migración en Django se refiere a los cambios que se hacen a través del framework a la estructura de la base de datos. Estos se hacen a través de los modelos en los archivos `models.py` en cada una de las aplicaciones que existan en el proyecto. En estos modelos es donde se define la estructura de las tablas y las relaciones a través del framework, utilizando un lenguaje Python y es Django quien hace las transformaciones.

¿Por qué son necesarias las migraciones en un proyecto Django?

Las migraciones son necesarias por varias razones:

En primer lugar, permiten un control de versiones de la base de datos manteniendo un registro ordenado de todos los cambios realizados en cada una de las migraciones, los cuales se guardan con un numero especifico como un archivo en la carpeta `migrations`.

Por otro lado, sincronizan de forma automática la estructura de la base de datos, lo que evita inconsistencias entre código y datos. Si es que hay alguna inconsistencia el framework levanta una alerta en la consola. Otro aspecto que evita inconsistencia y que a la vez facilita trabajo en equipo, es que las migraciones pueden ser replicadas a través de los archivos de migración, lo que permite que todo el equipo utilice el mismo esquema.

Finalmente, se destaca la portabilidad, ya que se pueden usar diferentes bases de datos y no es necesario utilizar el lenguaje especifico de cada sistema.

¿Qué problema resuelven y cómo facilitan el manejo de bases de datos?

Las migraciones resuelven varios problemas como es el evitar el uso de SQL manual cada vez que se cambia la estructura de la base de datos evitando errores. También, previene la perdida de datos al modificar la estructura ya que Django puede solicitar un valor por defecto para los registros existentes.

Por otro lado, al guardar cada migración en un archivo es posible volver a migraciones anteriores si es que es necesario o si se cometió un error en una migración más reciente y además aporta a la documentación del proyecto y su evolución.

2. Uso de Migraciones en Django

¿Cómo se generan automáticamente las migraciones en Django?

Se generan a través de comandos específicos que utilizan al archivo manage.py. A través de estos comandos que detecta los cambios realizados en los modelos y analiza las diferencias y luego genera un archivo nuevo que será destinado a la carpeta /migrations de cada aplicación al que se le asigna un numero secuencial, empezando desde el 0001 que dicta el orden en que se deben aplicar.

¿Cuál es el comando para crear una migración en Django? Explica su uso y da un ejemplo.

¿Cómo se aplican las migraciones nuevas en un proyecto Django?

A continuación, se contestan ambas preguntas, ya que tienen un hilo conductor entre si.

En primer lugar, se realiza `python manage.py makemigrations` para generar los archivos de las migraciones y luego se aplica `python manage.py migrate` para aplicar los cambios. Esto también se puede hacer para cada aplicación por separado si se prefiere realizar cambios de forma modular, para lo cual se debe agregar el nombre de la app al final del comando (ej. `python manage.py makemigrations nombre_app`).

En un caso de ejemplo, poniéndose en el caso de haber creado un modelo de nombre "Autor" en una app llamada "biblioteca". Si se quisiera hacer solo las migraciones de esta app, se debería utilizar "`python manage.py makemigrations biblioteca`" y Django entregaría un mensaje:

```
Migrations for 'biblioteca':

 biblioteca/migrations/0001_initial.py

+ Create model Autor
```

Lo que representa que se crea la migración. Después de esto, al ejecutar "`python manage.py migrate biblioteca`" se realizarán los cambios en la base de datos. Además, se puede utilizar "`python manage.py showmigrations`" para comprobar el historial de cambios.

¿Qué sucede si una migración no se aplica correctamente? ¿Cómo se puede solucionar?

Si no se aplica correctamente, puede darse debido a distintas razones dentro de los cuales se encuentran:

Error de integridad de datos: Ocurre cuando una migración intenta hacer cambios que violan restricciones existentes (por ejemplo, agregar un campo NOT NULL sin valor por defecto cuando ya hay datos).

Conflictos de migraciones: Sucede cuando dos desarrolladores crean migraciones con el mismo número o cuando hay cambios en ramas diferentes que afectan los mismos modelos.

Dependencias faltantes: Cuando una migración depende de otra que no ha sido aplicada o no existe.

Errores de sintaxis o lógica: Problemas en el código de la migración misma.

Y para cada uno de ellos se debe revisar el código en donde ocurra el error.

También existe una forma de revertir migraciones como "`python manage.py migrate biblioteca 0002`", en donde en este ejemplo especifico, se volvería a la migración numero 2 y se revierten todas las migraciones posteriores. Otro comando que realiza una función similar es "`python manage.py migrate biblioteca zero`" que deshace todas las migraciones de la app biblioteca.

3. Aplicación Práctica

Lee el siguiente fragmento de código y responde:

```python
from django.db import models

class Autor(models.Model):
    nombre = models.CharField(max_length=100)
    email = models.EmailField(unique=True)

class Libro(models.Model):
    titulo = models.CharField(max_length=200)
    autor = models.ForeignKey(Autor, on_delete=models.CASCADE)
    fecha_publicacion = models.DateField()
```

Si acabas de agregar estos modelos a un proyecto Django, ¿qué pasos debes seguir para asegurarte de que están reflejados en la base de datos?

Primero, asegurarse de que la aplicación donde definimos estos modelos esté incluida en INSTALLED_APPS dentro del archivo `settings.py`; Luego, revisar que los modelos estén bien definidos; Generar las migraciones con `python manage.py makemigrations`; Finalmente, utilizar `python manage.py migrate.`

¿Qué comando ejecutarías para crear y aplicar la migración correspondiente?

Como se mencionó anteriormente, se utilizaría `python manage.py makemigrations` para crear la migración y `python manage.py migrate` para aplicarla. O también, en su defecto `manage.py makemigrations nombre_app` y `python manage.py migrate nombre_app.`

Supongamos que decides agregar un nuevo campo `isbn` en el modelo `Libro` después de haber aplicado la primera migración. ¿Qué pasos debes seguir para actualizar la base de datos sin perder datos existentes?

Primero se debe agregar al modelo en la misma indentación que los campos `título, autor` y `fecha_publicacion` con una estructura como:

```
isbn = models.CharField(max_length=13, blank=True, null=True)
```

En la estructura anterior, se utilizan campos `blank` y `null` como True para que se permita que en los registros existentes este campo pueda estar vacío o también se le puede asignar un valor por defecto donde se debería cambiar `blank` y `true` por un `default='valor_default'.`

Después de agregar esto al modelo, se deberán ejecutar los comandos `makemigrations` y `migrate`.

4. Reflexión Final

¿Cuál crees que es la importancia de manejar correctamente las migraciones en un proyecto en equipo?

La importancia de utilizar de forma correcta las migraciones en un proyecto en equipo recae en poder tener una estructura de datos robusta y coherente para todos los miembros del equipo.

Al tener todos las mismas migraciones, se permite que la estructura de la base de datos sea dictada por la versión del proyecto y asi, todos trabajaran bajo la misma base. Lo anterior permite que se eviten errores como perdida de datos o errores en la integridad del proyecto. Además, como se menciono en la actividad, las migraciones permiten la documentación, lo que aporta a la trazabilidad y un control general del proyecto.

Todo lo que se menciona es una parte de toda la importancia que significa el uso de migraciones, pero son elementos que se consideran cruciales y siempre buscando el poder tener un proyecto coherente, limitar los errores y facilitar el flujo de trabajo del equipo completo.

¿Qué errores comunes pueden surgir al trabajar con migraciones y cómo evitarlos?

Dentro de los errores que se pueden encontrar están:

1. Modificar migraciones ya aplicadas

Problema: Editar un archivo de migración que ya fue aplicado en otros ambientes causa inconsistencias y conflictos de estado.

Cómo evitarlo: Nunca modificar migraciones ya compartidas con el equipo o aplicadas en producción. Si hay errores, crear una nueva migración correctiva.

2. Conflictos de migraciones en merge

Problema: Dos desarrolladores crean migraciones con el mismo número en ramas diferentes al trabajar en paralelo.

Cómo evitarlo: Mantener comunicación constante sobre cambios en modelos y usar `python manage.py makemigrations --merge` para resolver conflictos.

3. No incluir migraciones en el control de versiones

Problema: Agregar la carpeta migrations/ al `.gitignore` impide que otros desarrolladores obtengan las migraciones necesarias.

Cómo evitarlo: Siempre incluir las migraciones en Git. Solo excluir `__pycache__/` y archivos `.pyc`.

4. No probar migraciones antes de producción

Problema: Aplicar migraciones directamente en producción sin pruebas previas puede causar errores, pérdida de datos o downtime.

Cómo evitarlo: Establecer un pipeline (desarrollo → staging → producción), hacer backups antes de migrar y probar todas las migraciones en ambientes de prueba.

5. Eliminar campos sin considerar datos existentes

Problema: Eliminar campos con datos importantes resulta en pérdida permanente de información.

Cómo evitarlo: Verificar si hay datos importantes antes de eliminar. Implementar un proceso de dos pasos: primero hacer el campo opcional (null=True), luego eliminarlo después de migrar o respaldar los datos.

6. No ejecutar migrate después de makemigrations

Problema: Crear la migración, pero olvidar aplicarla causa desincronización entre el código y la base de datos.

Cómo evitarlo: Ejecutar ambos comandos en secuencia o usar un alias: `python manage.py makemigrations && python manage.py migrate`

Mejores prácticas generales:

- Hacer backups antes de migraciones importantes
- Revisar el código de las migraciones generadas automáticamente
- Comunicar cambios importantes en la base de datos al equipo
- Usar ambientes separados para probar cambios antes de producción
- Documentar migraciones complejas con nombres descriptivos