Programación 2 > módulo 3

>Tecnicatura Universitaria en Desarrollo de Software

UPATECO UNIVERSIDAD PROVINCIAL
ADMINISTRACIÓN • TECNOLOGÍA • OFICIOS

Ministerio de Economía
y Servicios Públicos
Gobierno de Salta

academia
CIMNE IBER

módulo 3

Tecnicatura Universitaria en Desarrollo de Software

Programación 2

Módulo 3: Backend

Tema 5: Manejo de errores

UPATECO UNIVERSIDAD PROVINCIAL ADMINISTRACIÓN • TECNOLOGÍA • OFICIOS

Ministerio de Economía y Servicios Públicos Gobierno de Salta

academia CIMNE IBER

Trabajo Práctico 3.3

El objeto de este trabajo es poner en práctica los conocimientos adquiridos en el módulo sobre manejo de errores y excepciones, y serialización de objetos.

Para ello se brindará una API con algunas rutas implementadas, y se solicitará implementar manejadores de errores para las mismas, además de realizar algunas modificaciones en el código para mejorar su legibilidad y mantenibilidad.

Se empleará la base de datos `sakila` de MySQL, particularmente la tabla `film` para obtener información de películas.

Por último, en clase se estableció que existen varias alternativas para definir excepciones personalizadas, como heredar de `HTTPException` de `werkzeug.exceptions`. En este trabajo se solicita definir excepciones personalizadas heradando de la siguiente clase:

```
class CustomException(Exception):

    def __init__(self, status_code, name = "Custom Error", description
= 'Error'):
        super().__init__()
        self.description = description
        self.name = name
        self.status_code = status_code
    def get_response(self):
        response = jsonify({
            'error': {
                'code': self.status_code,
                'name': self.name,
                'description': self.description,
            }
        })
        response.status_code = self.status_code
        return response
```

Aclaraciones

Veremos que el modelo `Film` cuenta con un método para serializar una instancia de esta clase como un diccionario y así poder devolver una respuesta con formato JSON. Este método no debe ser modificado, pero quisiéramos explicar ciertos detalles que pueden apreciarse en el mismo.

- Los atributos `rental_rate` y `replacement_cost` son de tipo `Decimal` en la base de datos, pero JSON no soporta este tipo de dato. Por lo tanto, se convierten a un número entero multiplicando por 100 para no perder precisión. Por ejemplo, si el precio del alquiler es 4.99, se convierte a 499.
- El atributo `last_update` es de tipo `datetime` en la base de datos, pero JSON no soporta este tipo de dato. Por lo tanto, se convierte a un string con el formato `YYYY-MM-DD HH:MM:SS`.
- El atributo `special_features` es de tipo `set` en la base de datos, pero JSON no soporta este tipo de dato. Por lo tanto, se convierte a una lista para una mejor representación, siempre y cuando no sea nulo.

Ejercicio 1

Al realizar una petición GET a la ruta `/films/<int:film_id>` se obtiene información de una película por su ID. Sin embargo, si el ID no existe, se producirá un error debido a que el controlador no está preparado para manejar esta situación. Es decir, no devolvemos una respuesta con un formato adecuado para el cliente.

Por lo tanto, se solicita implementar una excepción personalizada llamada `FilmNotFound` que herede de la clase `CustomException` y que se lance cuando no se encuentre una película por su ID.

Por supuesto, se debe definir un manejador para esta excepción, y registrar el manejador en la aplicación.

Una vez completado lo anterior, este deberá devolver una respuesta con formato JSON y un código de estado HTTP `404`. El cuerpo de la respuesta debe tener el siguiente formato:

```
{
    "error": {
        "code": 404,
        "name": "Film Not Found",
        "description": "Film with id {film_id} not found"
    }
}
```

Ejercicio 2

Al registrar una nueva película con un método POST para el *endpoint* `/films/` algunos datos son **obligatorios**, como el id de la película, título, id del idioma, duración del alquiler, precio del alquiler, costo de reemplazo y última actualización.

Los datos son ingresados por el usuario, por lo tanto, se solicita validar los datos de entrada:

- El atributo `title` debe tener tres caracteres como mínimo.
- Los atributos `language_id` y `rental_duration` deben ser números enteros.
- Como hemos mencionado, los atributos `rental_rate` y `replacement_cost` son de tipo `Decimal` en la base de datos y los convertimos a un número entero multiplicando por 100 al momento de serializar la película. No obstante, no existe una validación para estos campos al momento de registrar una nueva película. Por lo tanto, se solicita validar que estos campos sean números enteros.
- En cuanto al atributo `last_update` y el `id` de la película, estos son generados automáticamente por la base de datos y en la misma consulta de inserción. Por lo tanto, no deben ser ingresados por el usuario, pues serán ignorados.

Por último, para el atributo `special_features` se solicita validar que el valor ingresado sea una lista de strings, y que cada string sea uno de los siguientes: `Trailers`, `Commentaries`, `Deleted Scenes`, `Behind the Scenes`.

> Ayuda: para validar que un valor sea una lista de strings, se puede emplear la función `isinstance` de Python. Por ejemplo, `isinstance(data.get('special_features'), list)` devuelve `True` si el valor de `special_features` es una lista, o `False` en caso contrario.

> Recordemos que debemos validar también que cada string sea uno de los mencionados anteriormente.

Si no se cumplen las validaciones mencionadas, se debe lanzar una excepción personalizada llamada `InvalidDataError` con un código de estado HTTP `400`, la cual hereda de la clase `CustomException`. Por supuesto, se debe definir un manejador para esta excepción, y registrar el manejador en la aplicación.

Ejercicio 3

Al realizar una petición PUT a la ruta `/films/<int:film_id>` se actualiza una película por su ID. Sin embargo, vemos que si este ID existe o no, la respuesta es la misma. Esto se debe a que no se está validando si el ID existe antes de intentar actualizar la película.

Para solucionar esto, se solicita implementar un método de instancia en el modelo `Film` llamado `exists` que devuelva `True` si el ID de la película existe en la base de datos, o `False` en caso contrario.

```python
def exists(self):
    # Implementar
```

Una vez definido el método, se debe modificar el método de clase `update` del **controlador** empleando el método `exists` mencionado anteriormente. Si el ID no existe, se debe lanzar una excepción personalizada llamada `FilmNotFound`, la cual hemos definido en el ejercicio 1.

Ejercicio 4

Al realizar una petición PUT a la ruta `/films/<int:film_id>` se actualiza una película por su ID. No obstante, tal y como está implementado el controlador, podrían enviar un JSON con un formato incorrecto, y esto podría generar un error en el servidor.

Por lo tanto, de igual manera que en el ejercicio 2, se solicita validar los datos de entrada:

- El atributo `title` debe tener tres caracteres como mínimo.
- Los atributos `language_id` y `rental_duration` deben ser números enteros.
- Los atributos `rental_rate` y `replacement_cost` deben ser números enteros.

- En cuanto al atributo `special_features` se solicita validar que el valor ingresado sea una lista de strings, y que cada string sea uno de los siguientes: `Trailers`, `Commentaries`, `Deleted Scenes`, `Behind the Scenes`.

Si no se cumplen las validaciones mencionadas, se debe lanzar una excepción personalizada llamada `InvalidDataError`, la cual definimos en el ejercicio 2.

Ejercicio 5

Al realizar una petición DELETE a la ruta `/films/<int:film_id>` se elimina una película por su ID. Sin embargo, sin importar si eliminamos una película o no, la respuesta es la misma. Esto se debe a que no se está validando si el ID existe antes de intentar eliminar la película.

De igual manera que en el ejercicio 3, se solicita emplear el método `exists` solicitado en el ejercicio antes mencionado para validar si el ID de la película existe en la base de datos. Si el ID no existe, se debe lanzar una excepción personalizada llamada `FilmNotFound`, la cual definimos en el ejercicio 1.