



# Django REST Framework - Creating a Booking API

- [Django REST Framework - Creating a Booking API](#)
  - [Serializers](#)
  - [Views](#)
  - [URL Configuration](#)
  - [Concurrency](#)
    - [Soft Locking](#)
      - [Soft Locking in BookMyShow](#)
    - [Transactions in Django REST Framework:](#)
    - [Implementing `create\_booking` Method as Transactional:](#)
  - [Calculating the Booking Amount](#)
    - [Testing the API](#)
    - [Conclusion](#)

In the BookMyShow application, creating a booking is the most critical part of the system. It involves several steps, including user validation, seat availability checks, transaction management, and pricing calculation. In this document, we'll discuss the Django REST Framework implementation for the booking creation API, focusing on transactions and pricing strategies.

Before we get started, let us list the process for implementing the booking creation API:

1. Create a serializer for the `Booking` model
2. Implement the booking creation view
3. Modify the URL configuration

## Serializers

First, let's create a serializer for the `Booking` model. In the `serializers.py` file in the `bookings` app, add the following code:

```
from rest_framework import serializers
from .models import Booking

class BookingSerializer(serializers.ModelSerializer):
    class Meta:
        model = Booking
        fields = '__all__'
```

This serializer will handle the conversion of `Booking` model instances to JSON and vice versa.

## Views

Now, let's create views for handling the booking creation API.

The logical steps for creating a booking are:

1. Get the user
2. Get the show
3. Get the show seats
4. Check if all the seats are available
5. Mark all the seats as locked
6. Calculate and update the booking amount
7. Create and save the booking

In the `views.py` file in the `bookings` app, add the following code:

```

from rest_framework import generics, status
from rest_framework.response import Response
from django.shortcuts import get_object_or_404
from .models import Booking, ShowSeat, SeatStatus
from .serializers import BookingSerializer
from .services import PricingStrategy

class BookingCreateView(generics.CreateAPIView):
    serializer_class = BookingSerializer

    def create(self, request, *args, **kwargs):
        # Step 1 – Get the user through ID
        # Step 1b) – If user is not present, throw error
        user_id = request.data.get('user_id')
        user = get_object_or_404(User, id=user_id)

        # Step 2 – Get the show using show ID
        # Step 2b) – If show is not present, throw error
        show_id = request.data.get('show_id')
        show = get_object_or_404>Show, id=show_id)

        # Step 3 – Get the show seats using showSeat IDs
        # Step 4 – Check if all the seats are available
        show_seat_ids = request.data.get('seat_ids', [])
        show_seats = ShowSeat.objects.filter(id__in=show_seat_ids)

        for seat in show_seats:
            if seat.status != SeatStatus.AVAILABLE:
                raise ValueError("Seat is not available")

        # Step 5 – Mark all the seats as locked
        for seat in show_seats:
            seat.status = SeatStatus.LOCKED
            seat.save()

        # Step 7 – Create and save the booking
        booking_data = {
            "user": user.id,
            "show": show.id,

```

```

        "seats": show_seats,
        "status": BookingStatus.PENDING,
        "booked_at": timezone.now(),
    }

    serializer = BookingSerializer(data=booking_data)
    serializer.is_valid(raise_exception=True)
    serializer.save()

    # Step 8 - Calculate and update the booking amount
    # We shall come back to this step later
    amount = None
    serializer.instance.amount = amount
    serializer.instance.save()

    return Response(serializer.data, status=status.HTTP_201_CREATED)

```

In this view, we've created a `BookingCreateView` class that inherits from `CreateAPIView`. This view handles the `POST` requests for creating a new booking. There are a lot of things going on in this method, so let's break it down into steps:

1. `Fetching a user and a show` - The first step is to fetch the user and the show using their IDs. If the user or the show is not present, we throw an error. This can be done using the `get_object_or_404` method which returns the object if it exists, or throws a `Http404` exception.
2. `Fetching show seats` - The next step is to fetch the show seats using their IDs. You can query the database using the models you created in the previous sessions. Here, we only want the seats with the ids that are passed in the request. We can do this using the filter method and the `__in` operator. This will write a query like `SELECT * FROM show_seat WHERE id IN (1, 2, 3, 4, 5)`.
3. `Checking seat availability` - Once we have the show seats, we need to check if all the seats are available. If any of the seats are not available, we throw an error. This can be done by iterating over the show seats and checking if the status is `SeatStatus.AVAILABLE`. If not, we raise a `ValueError`.
4. `Locking the seats` - Once we've checked the seat availability, we need to lock the seats. This can be done by iterating over the show seats and setting the status to `SeatStatus.LOCKED`. We also need to save the seats after updating the

status.

5. **Creating the booking** - Now that we have the user, show, and show seats, we can create the booking. We create a dictionary with the booking data and pass it to the `BookingSerializer`. We then check if the serializer is valid and save the booking. The `save` method will perform the database operation to create the booking.

## HTTP Status Codes

HTTP status codes are a standard way of communicating the status of an HTTP request. There are several status codes, and each code has a specific meaning.

Following are the different families of status codes: - 1xx - Informational

- 2xx - Success
- 3xx - Redirection
- 4xx - Client Error
- 5xx - Server Error

Some common status codes are:

- 200 - OK - The request was successful
- 201 - Created - The request was successful and a new resource was created
- 400 - Bad Request - The request was invalid and could not be processed
- 401 - Unauthorized - The request was not authorized
- 404 - Not Found - The requested resource was not found
- 500 - Internal Server Error - The server encountered an error while processing the request

## URL Configuration

Register the URL for the booking creation view in the `urls.py` file in the `bookings` app:

```
from django.urls import path
from .views import BookingCreateView

urlpatterns = [
    path('booking/', BookingCreateView.as_view(), name='booking-create'),
]
```

This sets up an endpoint `/bookings/create/` for creating a new booking.

## Concurrency

A shortcoming of the above implementation is that it does not handle concurrency. If two users try to book the same seat at the same time, both the bookings will be created. This is because the seat availability check and the seat locking are not atomic operations. To solve this problem, we can use different mutual exclusion or locking techniques.

## Soft Locking

Soft locking is a concurrency control mechanism used to manage access to shared resources, ensuring that multiple transactions do not interfere with each other. In the context of a ticket booking system like BookMyShow, soft locking is often employed to prevent multiple users from simultaneously booking the same seat for a show.

The basic idea behind soft locking is to mark a resource (e.g., a seat) as temporarily unavailable or "locked" when a user starts a transaction to reserve or book that resource. This prevents other users from accessing or modifying the same resource concurrently until the transaction is completed. Soft locking is "soft" because it relies on cooperation between transactions rather than rigid locks that block access completely.

## Soft Locking in BookMyShow

In the context of BookMyShow, the process might look like this:

### 1. User A selects seats for booking:

- The seats are marked as "soft-locked" to indicate that User A is in the process of booking them.
- These soft-locked seats are temporarily unavailable to other users.

## 2. User B attempts to select the same seats:

- Since the seats are soft-locked, User B is notified that the seats are currently being booked by another user.
- User B can then choose alternative seats or wait for the soft-lock to be released.

## 3. User A completes the booking transaction:

- The soft lock is released, and the seats are marked as officially booked.
- Other users can now see that these seats are no longer available.

# Transactions in Django REST Framework:

Django provides built-in support for database transactions, and Django REST Framework (DRF) inherits this functionality. In DRF, you can use the `@transaction.atomic` decorator or the `transaction.atomic` context manager to ensure that a block of code is executed within a single database transaction.

## Implementing `create_booking` Method as Transactional:

Assuming you have a `BookingService` class with a `create_booking` method, you can make it transactional using the `@transaction.atomic` decorator. However, we don't need to make the entire method transactional. We only need to make the seat locking part transactional. So, you can extract the seat locking logic into a separate method and make it transactional.

We will use the `serializable` isolation level, which is the highest level of isolation and ensures that the transaction is executed in a serial order, and only one transaction can access a resource at a time.

```

from django.db import transaction
from django.utils import timezone
from .models import Booking, ShowSeat, SeatStatus

class BookingService:
    @transaction.atomic(isolation=transaction.ISOLATION_LEVEL_SERIALIZABLE)
    def lock_seats(self, show_seats):
        show_seat_ids = request.data.get('seat_ids', [])
        show_seats = ShowSeat.objects.filter(id__in=show_seat_ids)

        for seat in show_seats:
            if seat.status != SeatStatus.AVAILABLE:
                raise ValueError("Seat is not available")

        # Step 5 – Mark all the seats as locked
        for seat in show_seats:
            seat.status = SeatStatus.LOCKED
            seat.save()

```

Here, the `@transaction.atomic` decorator ensures that the entire `lock_seats` method is executed within a single database transaction. If any exception occurs during the process, the transaction is rolled back, and the database is left in a consistent state.

This approach helps maintain the integrity of the database and prevents issues related to concurrent access to the same resources, providing a level of soft locking within the transactional context. It ensures that either the entire booking process succeeds, or it fails and leaves the system in a consistent state.

## Calculating the Booking Amount

The booking amount is calculated based on the number of seats booked and the price of each seat. However, this is not as simple as it sounds. It could be based on multiple factors such as:

- Time of the show
- Day of the week
- Seat type



- Theatres

To handle these different pricing strategies, we can use the Strategy design pattern. The Strategy pattern is a behavioral design pattern that enables selecting an algorithm at runtime. It defines a family of algorithms, encapsulates each algorithm, and makes the algorithms interchangeable within that family.

The `PricingStrategy` class is the base class for all pricing strategies. It defines an interface for calculating the price. The `calculate_price` method is implemented by the concrete classes that inherit from the `PricingStrategy` class.

```
from abc import ABC, abstractmethod

class PricingStrategy(ABC):
    @abstractmethod
    def calculate_price(self, booking: Booking) -> float:
        pass
```

Let us implement a simple pricing strategy just based on the number of seats booked and the type of seats.

```
class SeatBasedPricingStrategy(PricingStrategy):
    def calculate_price(self, booking: Booking) -> float:
        price = 0
        for seat in booking.seats.all():
            price += self.decide_price(seat.type)
        return price

    @staticmethod
    def decide_price(type: SeatType) -> float:
        return {
            SeatType.GOLD: 100,
            SeatType.PREMIUM: 200,
            SeatType.EXECUTIVE: 300,
        }[type]
```

## Testing the API

You can start the server using `python manage.py runserver` and test the API using Postman.

The API will be available at `http://localhost:8000/booking/`. You can send a `POST` request to this URL with the following body:

```
{
  "user_id": 1,
  "show_id": 1,
  "seat_ids": [1, 2, 3, 4, 5]
}
```

To create a booking, you would need to create the associated entities.


## Conclusion

In this session, we implemented the booking creation API using Django REST Framework, focusing on transactions, pricing strategies, and user validation. The API is designed to handle the booking creation process efficiently with proper error handling and response messages.

Next steps:

1. Create the associated views for the booking creation API such as `Movie`, `Show`, etc.
2. Implement the cancellation flow for the booking creation API.
3. Implement different pricing strategies based on the time of the show, day of the week, seat type, etc.



 You can find the associated code [here](#).