PROBLEM STATEMENT:

You have to design a REST interface for a movie theatre ticket booking system. It should support the following business cases:

● An endpoint to book a ticket using a user's name, phone number, and timings.

● An endpoint to update a ticket timing.

● An endpoint to view all the tickets for a particular time.

● An endpoint to delete a particular ticket.

● An endpoint to view the user's details based on the ticket id.

● Mark a ticket as expired if there is a diff of 8 hours between the ticket timing and current time.

● Note: For a particular timing, a maximum of 20 tickets can be booked.

● You should follow the REST paradigm while building your application.

● You can use any database you like.

● Create a proper readme for your project.

● Plus point if you could delete all the tickets which expired automatically.

● Plus point if you could write the tests for all the endpoints.

● Please attach a screenshot of your postman while testing your application.

● Please avoid plagiarism.

Solution:

I have used Laravel and MySQL to address this problem statement.

Laravel is an MVC framework and the directory structure of Laravel is such that

- In App/Http/Controller you would find the controllers.
- In App/Http there are the models
- In resources/views there are the views(not many in this project due to the backend necessity).
- The routes of all the APIs are in the Routes/api.php .

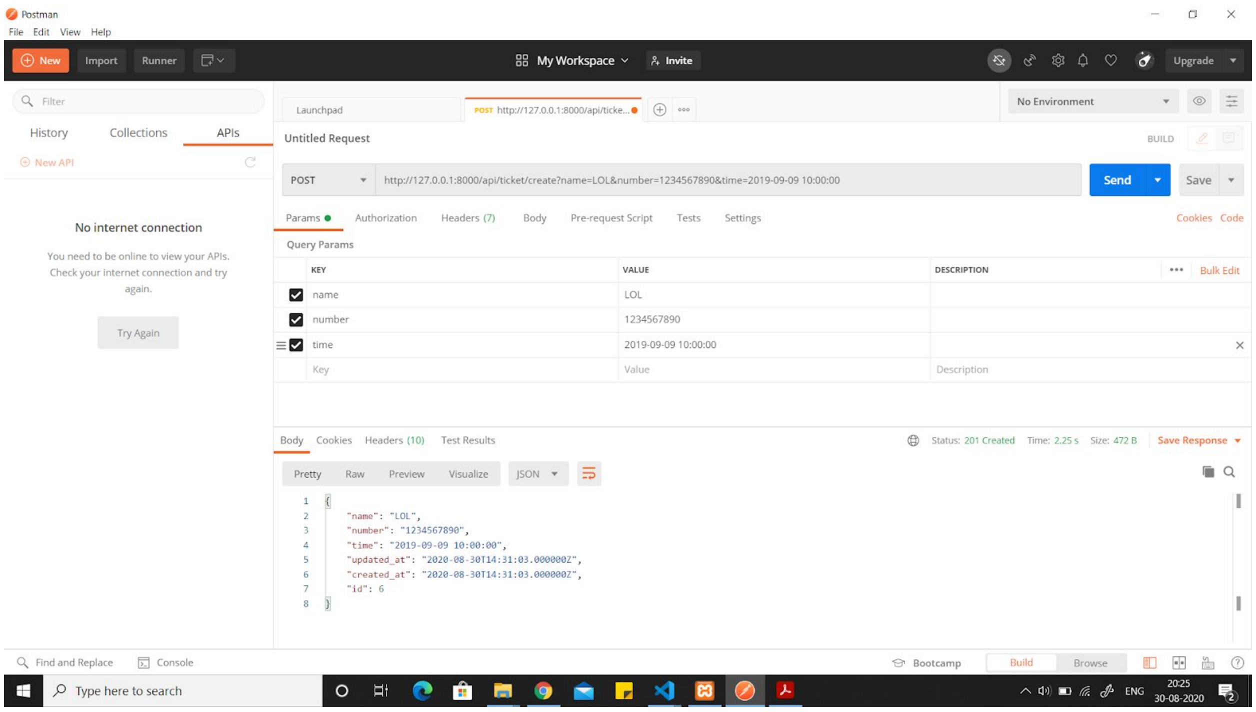
__Models:__

1. User: User table in MySQL to store the details of users that can use the functionality of the application after logging in.
2. Ticket: Ticket table to store all the information about the ticket for different API calling and storing purposes.

In order to test the API, you must login to the website by clicking login on the page or by navigating to localhost/login. You can register yourself or you can use the credentials:

Username/Email : 1234@1234.com
Password: 12345678

1. Now, according to the problem statement the first API was to book a ticket with appropriate fields:



Untitled Request

```
POST http://127.0.0.1:8000/api/ticket/create?name=LOL&number=1234567890&time=2019-09-09 10:00:00
```

Query Params

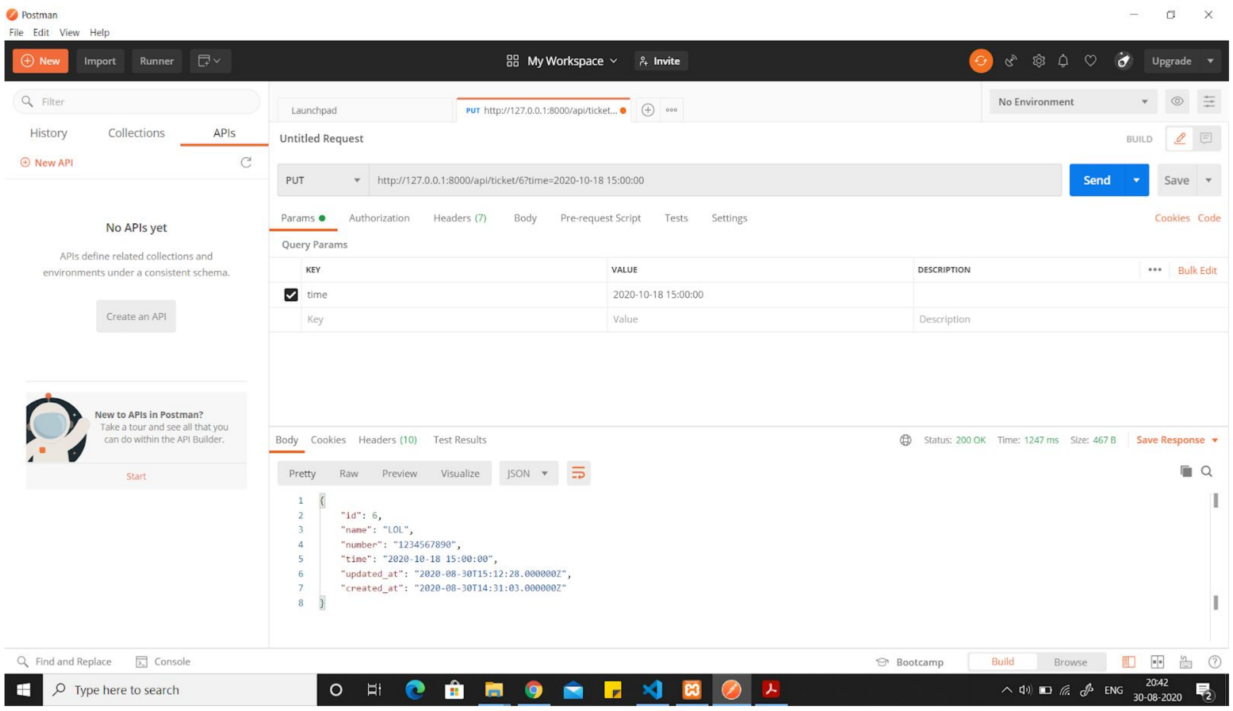
	KEY	VALUE	DESCRIPTION
✓	name	LOL	
✓	number	1234567890	
✓	time	2019-09-09 10:00:00	
	Key	Value	Description

Body — Status: 201 Created Time: 2.25 s Size: 472 B

```
{
    "name": "LOL",
    "number": "1234567890",
    "time": "2019-09-09 10:00:00",
    "updated_at": "2020-08-30T14:31:03.000000Z",
    "created_at": "2020-08-30T14:31:03.000000Z",
    "id": 6
}
```

The above screenshot shows the *POST* request that is being sent to the API ***localhost/api/ticket/create*** using the appropriate parameters and you can also see the response code along with its JSON values in the bottom half of the screen.

2. Second problem is to update the ticket timing.


PUT http://127.0.0.1:8000/api/ticket/6?time=2020-10-18 15:00:00

Query Params

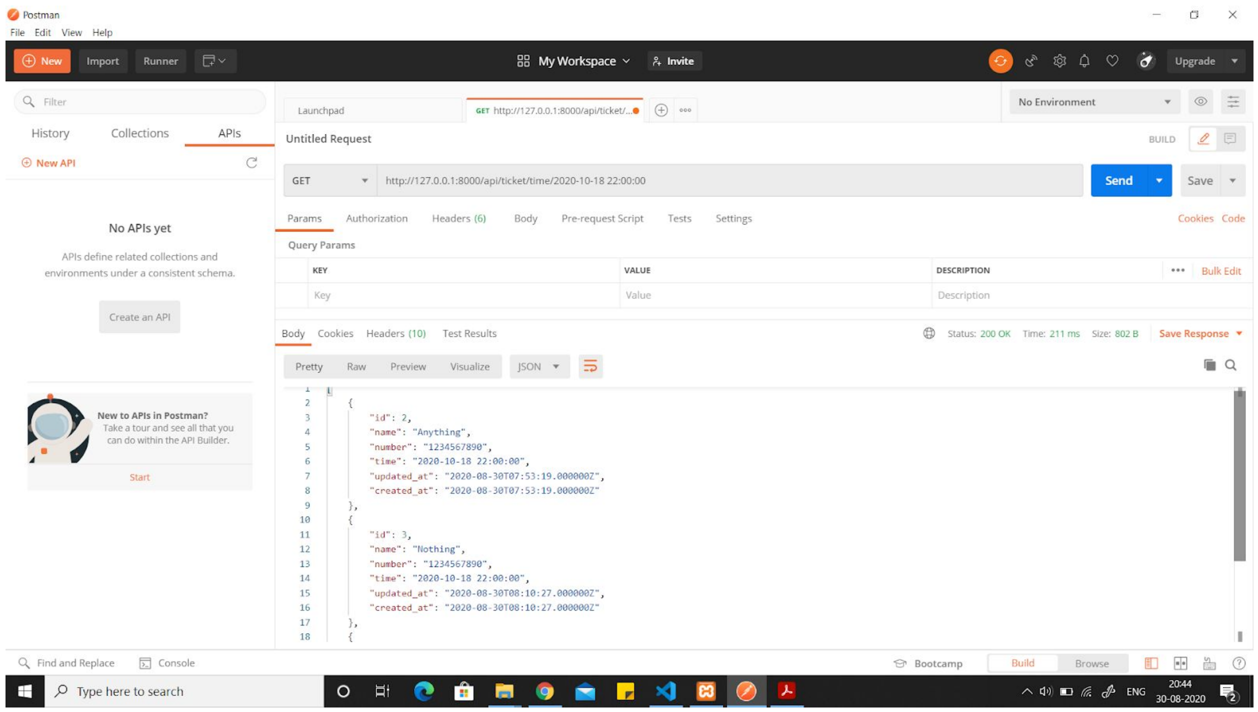
	KEY	VALUE	DESCRIPTION
✓	time	2020-10-18 15:00:00	

Status: 200 OK Time: 1247 ms Size: 467 B

```
{
    "id": 6,
    "name": "LOL",
    "number": "1234567890",
    "time": "2020-10-18 15:00:00",
    "updated_at": "2020-08-30T15:12:28.000000Z",
    "created_at": "2020-08-30T14:31:03.000000Z"
}
```

The above screenshot shows the *PUT* request that is being sent to the API ***localhost/api/ticket/{ticket}*** using the appropriate parameters and you can also see the response code along with its JSON values in the bottom half of the screen.

3. Next is to view all the tickets of a particular time



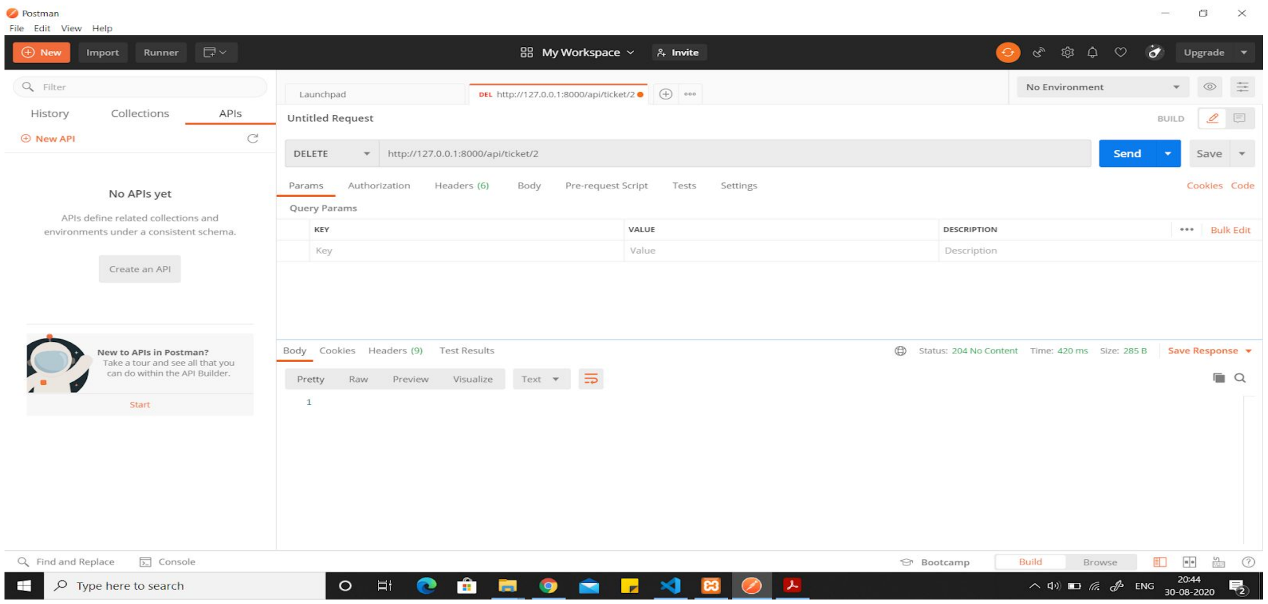
GET http://127.0.0.1:8000/api/ticket/time/2020-10-18 22:00:00

Status: 200 OK Time: 211 ms Size: 802 B

```
[
    {
        "id": 2,
        "name": "Anything",
        "number": "1234567890",
        "time": "2020-10-18 22:00:00",
        "updated_at": "2020-08-30T07:53:19.000000Z",
        "created_at": "2020-08-30T07:53:19.000000Z"
    },
    {
        "id": 3,
        "name": "Nothing",
        "number": "1234567890",
        "time": "2020-10-18 22:00:00",
        "updated_at": "2020-08-30T08:10:27.000000Z",
        "created_at": "2020-08-30T08:10:27.000000Z"
    },
    {
```

The above screenshot shows the *GET* request that is being sent to the API ***localhost/api/ticket/time/{time}*** using the appropriate parameters and you can also see the response code along with its JSON values that is the list of tickets booked for that time in the bottom half of the screen.

4. Next is to delete a ticket with a given ID.



Untitled Request

DELETE http://127.0.0.1:8000/api/ticket/2

Send Save

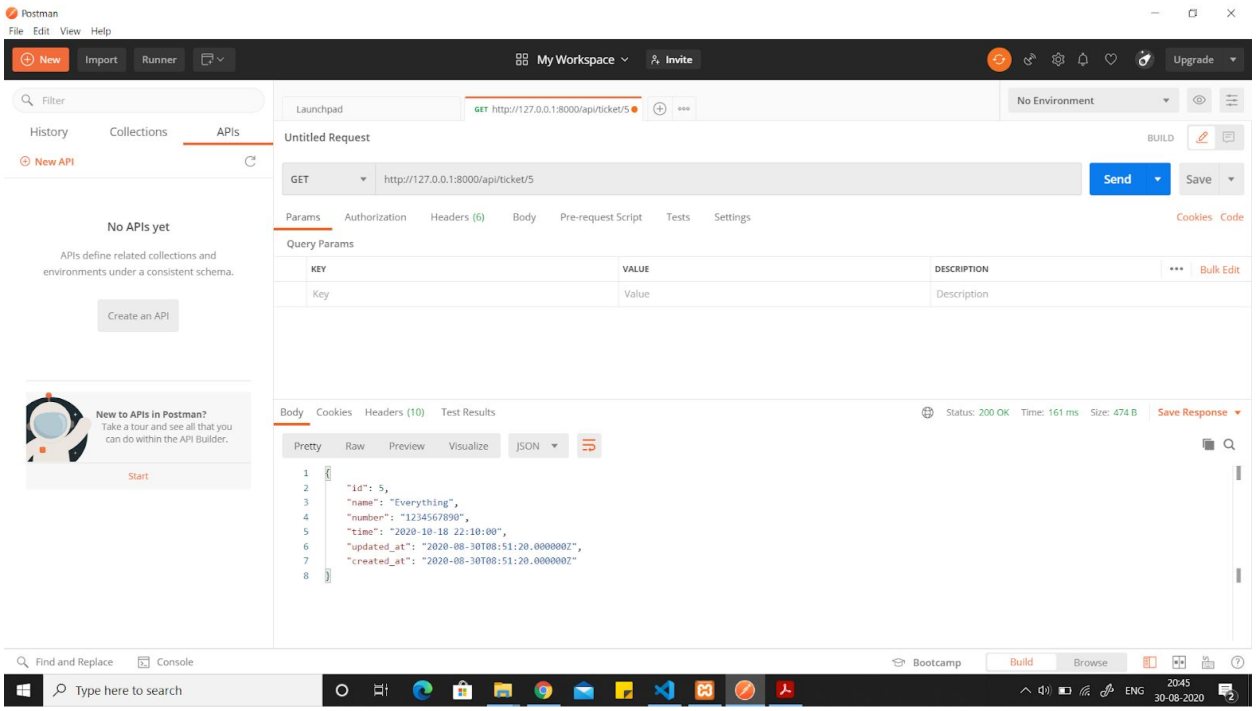
Status: 204 No Content Time: 420 ms Size: 285 B

```
1
```

20:44 30-08-2020

The above screenshot shows the *DELETE* request that is being sent to the API ***localhost/api/ticket/{ticket}*** using the appropriate parameters and you can also see the response code in the bottom half of the screen

5. Next is to retrieve all the information about the user who books the ticket using ticket id:



Untitled Request

GET http://127.0.0.1:8000/api/ticket/5

Send Save

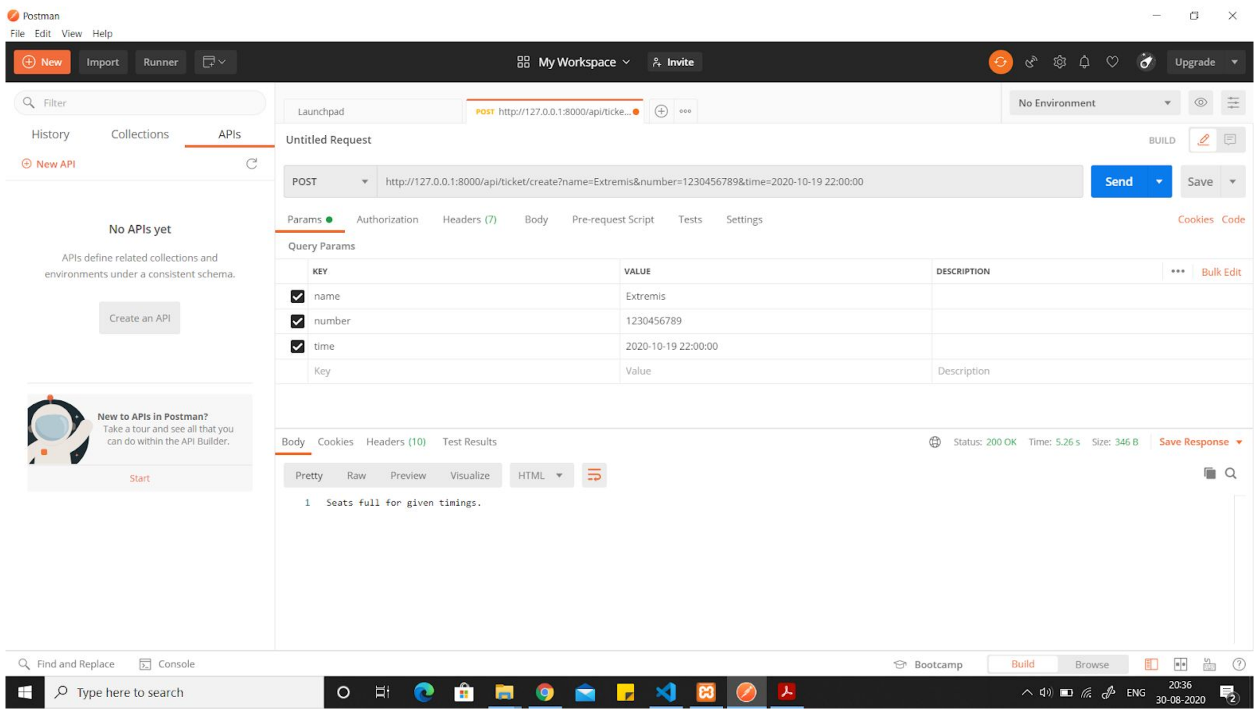
Status: 200 OK Time: 161 ms Size: 474 B

```
{
    "id": 5,
    "name": "Everything",
    "number": "1234567890",
    "time": "2020-10-18 22:10:00",
    "updated_at": "2020-08-30T08:51:20.000000Z",
    "created_at": "2020-08-30T08:51:20.000000Z"
}
```

20:45 30-08-2020

The above screenshot shows the *GET* request that is being sent to the API ***localhost/api/ticket/{id}*** using the appropriate parameters and you can also see the response code along with its JSON values that is the details of the person who booked that ticket in the bottom half of the screen.

6. One of the constraints was that the maximum number of tickets that can be booked for one time slot should be 20.



Untitled Request

```
POST http://127.0.0.1:8000/api/ticket/create?name=Extremis&number=1230456789&time=2020-10-19 22:00:00
```

Query Params

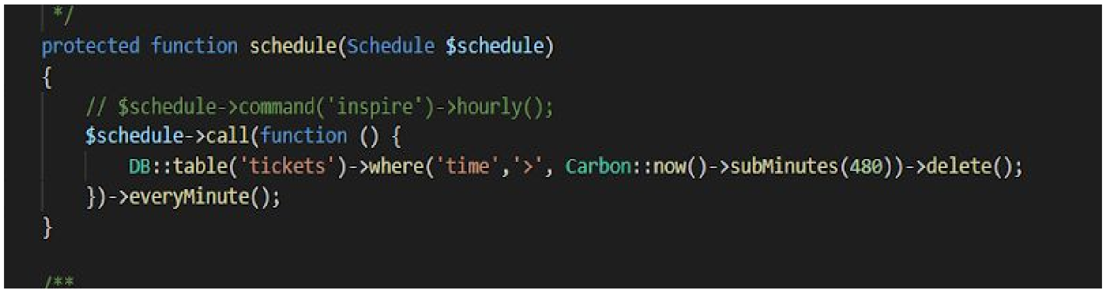
KEY	VALUE	DESCRIPTION
name	Extremis	
number	1230456789	
time	2020-10-19 22:00:00	

Status: 200 OK Time: 5.26 s Size: 346 B

```
Seats full for given timings.
```

Here, a request was sent to ‘create a ticket’ API (mentioned above) when the number of tickets that were in the database with required time was 20. In that case the API sends back a message that “***<u>Seats are full for given timings</u>***” which fulfills the above need.

7. One plus point was to delete tickets that were older than 8 hours from the current time(I have not used the status of expired because I am already deleting the expired tickets so adding another field would be a wastage of memory). However, I have used a concept known as scheduler from Laravel library that runs every minute and check if there is any entry with time less than 480 minutes(8 hours) of the current time and if it finds any, it deletes the entry automatically, Here is the code snippet of the above mentioned scheduler. You can find it in *App/Console/Kernel.*



```
 */
protected function schedule(Schedule $schedule)
{
    // $schedule->command('inspire')->hourly();
    $schedule->call(function () {
        DB::table('tickets')->where('time','>', Carbon::now()->subMinutes(480))->delete();
    })->everyMinute();
}

/**
```

8. I have also created some of the test cases for all APIs which are there in the *tests/Feature* folder.