Documentation: Payment Gateway Dotnet Core Challenge

Table of Contents

Application Description

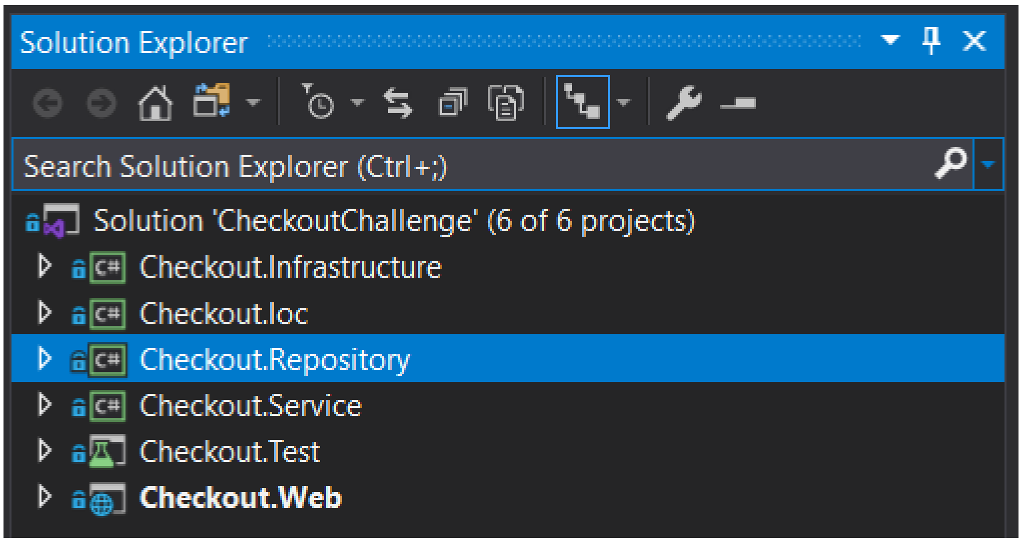
The application created is done using the .Net Core framework. A web application and web API have been implemented to allow merchants to process and retrieve payments. The API allows a merchant to retrieve all previously made payments through the payment gateway and displays it in a list for easy analysis, furthermore, a merchant can view the details of specific payment transaction based on the identifier of the previously made payment. Finally, a merchant can process a payment through the payment gateway with the appropriate HTTP responses to allow the merchant to know what was wrong during the process to make the payment.

Implementation of API

The payment gateway application was built using the onion architecture. The onion architecture was chosen as it is easier to make changes to the codes of the application as the application logics are separated from the UI layer. Most importantly, it is easier to debug the application without having to go through several lines of codes as the programmer will already know where the error is coming from and can go directly to that directory.

Application Architecture

The different layers of the onion architecture used in this application is shown in figure 1.



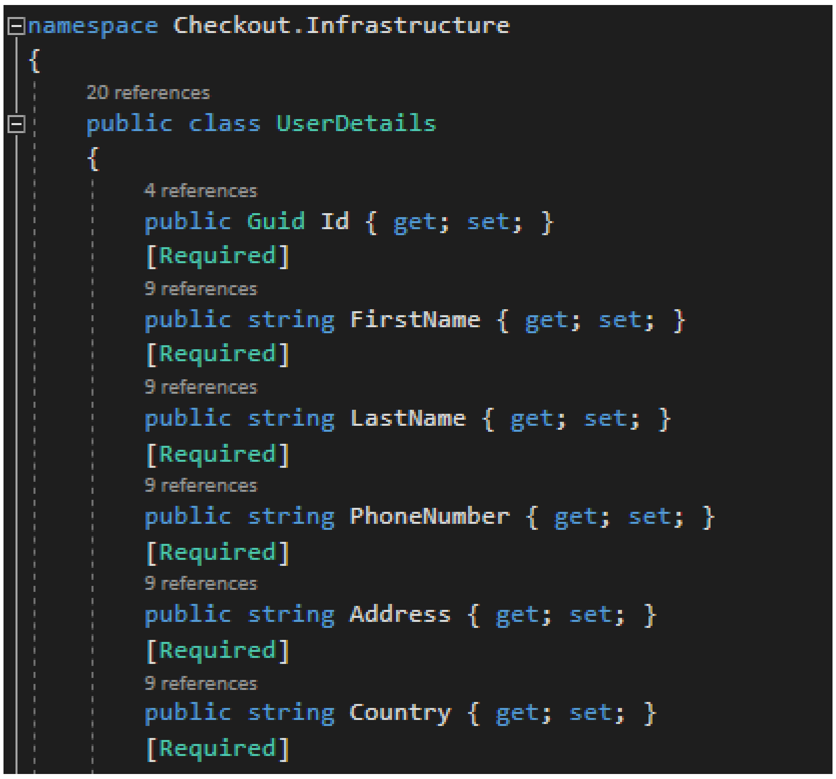
Solution 'CheckoutChallenge' (6 of 6 projects)
Checkout.Infrastructure
Checkout.Ioc
Checkout.Repository
Checkout.Service
Checkout.Test
Checkout.Web

Figure 1. Onion Architecture Project Layers

As it can be seen from figure 1, the application architecture is broken down into 6 different projects, this helps to reduce the lines of codes that needs to be written as the projects are dependencies of each other and the objects can be used in between the different projects successfully.

Checkout.Infrastructure Layer

The first project in the application is a class library named “Checkout.Infrastructure”, this layer represents the business and behavior objects. The idea of this layer is to have all the domain objects which then can be used throughout the other projects by referencing the class library. Figure 2 shows several lines of codes found in “Checkout.Infrastructure” class library.

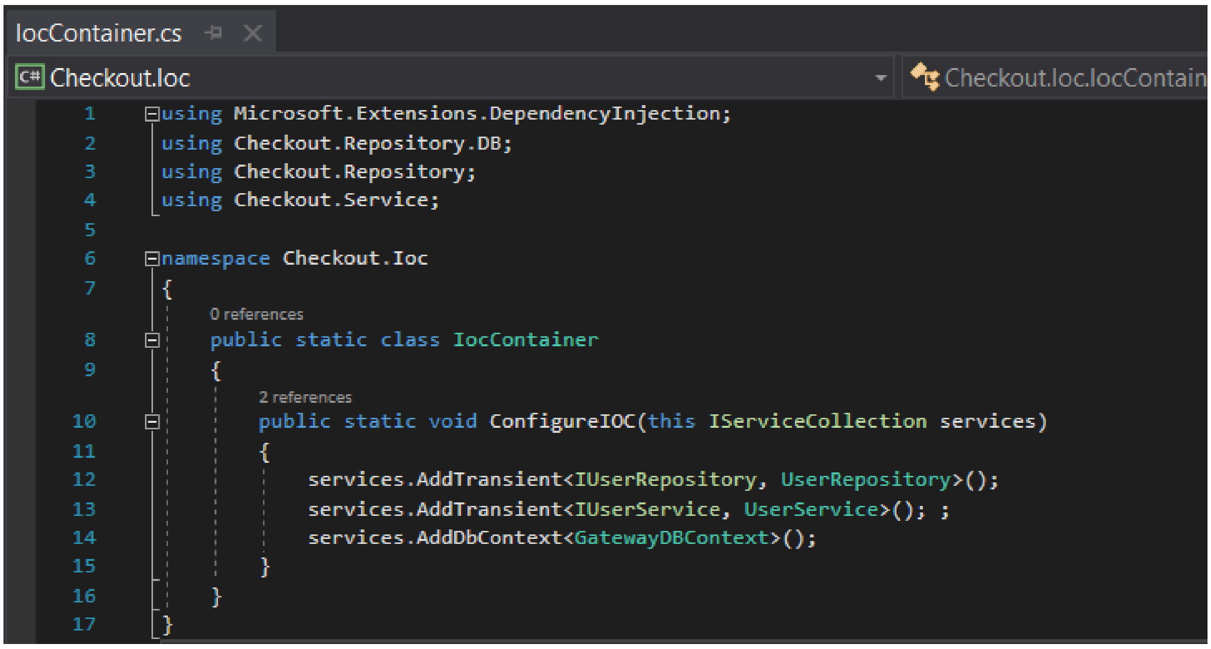


```
namespace Checkout.Infrastructure
{
    20 references
    public class UserDetails
    {
        4 references
        public Guid Id { get; set; }
        [Required]
        9 references
        public string FirstName { get; set; }
        [Required]
        9 references
        public string LastName { get; set; }
        [Required]
        9 references
        public string PhoneNumber { get; set; }
        [Required]
        9 references
        public string Address { get; set; }
        [Required]
        9 references
        public string Country { get; set; }
        [Required]
```

Figure 2. Checkout.Infrastructure Code Snippet

Checkout.Ioc Layer

Secondly, the class library named “Checkout.Ioc” is for the IoC container of the application, which is used to register the application services of the program. Figure 3 shows code snippet from the class library, it shows how the different services in the application are registered with a lifetime.

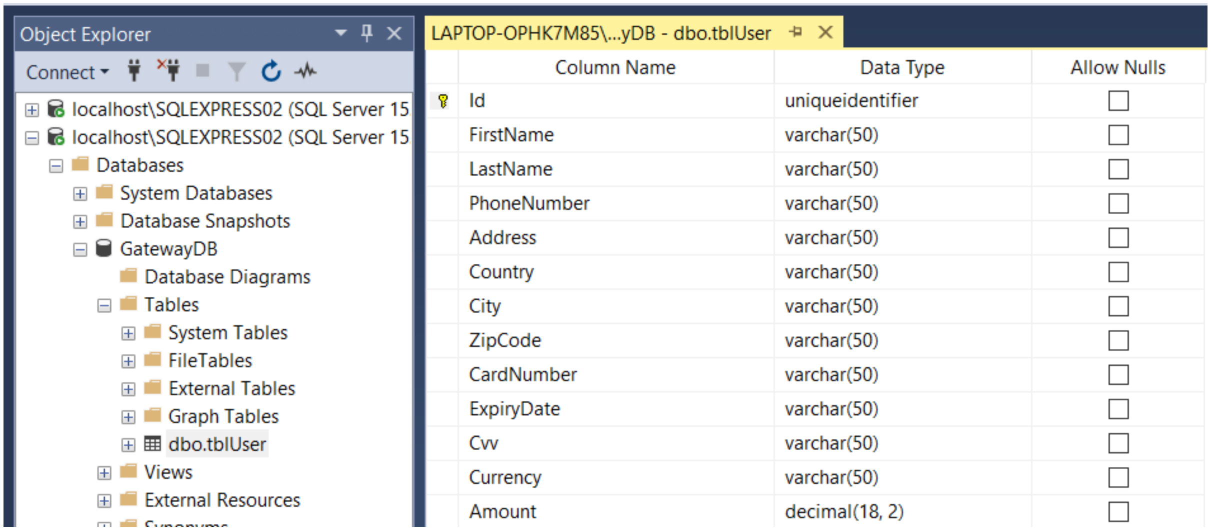


```
using Microsoft.Extensions.DependencyInjection;
using Checkout.Repository.DB;
using Checkout.Repository;
using Checkout.Service;

namespace Checkout.Ioc
{
    0 references
    public static class IocContainer
    {
        2 references
        public static void ConfigureIOC(this IServiceCollection services)
        {
            services.AddTransient<IUserRepository, UserRepository>();
            services.AddTransient<IUserService, UserService>(); ;
            services.AddDbContext<GatewayDBContext>();
        }
    }
}
```

Figure 3. Checkout.Ioc Code Snippet

Checkout.Repository Layer

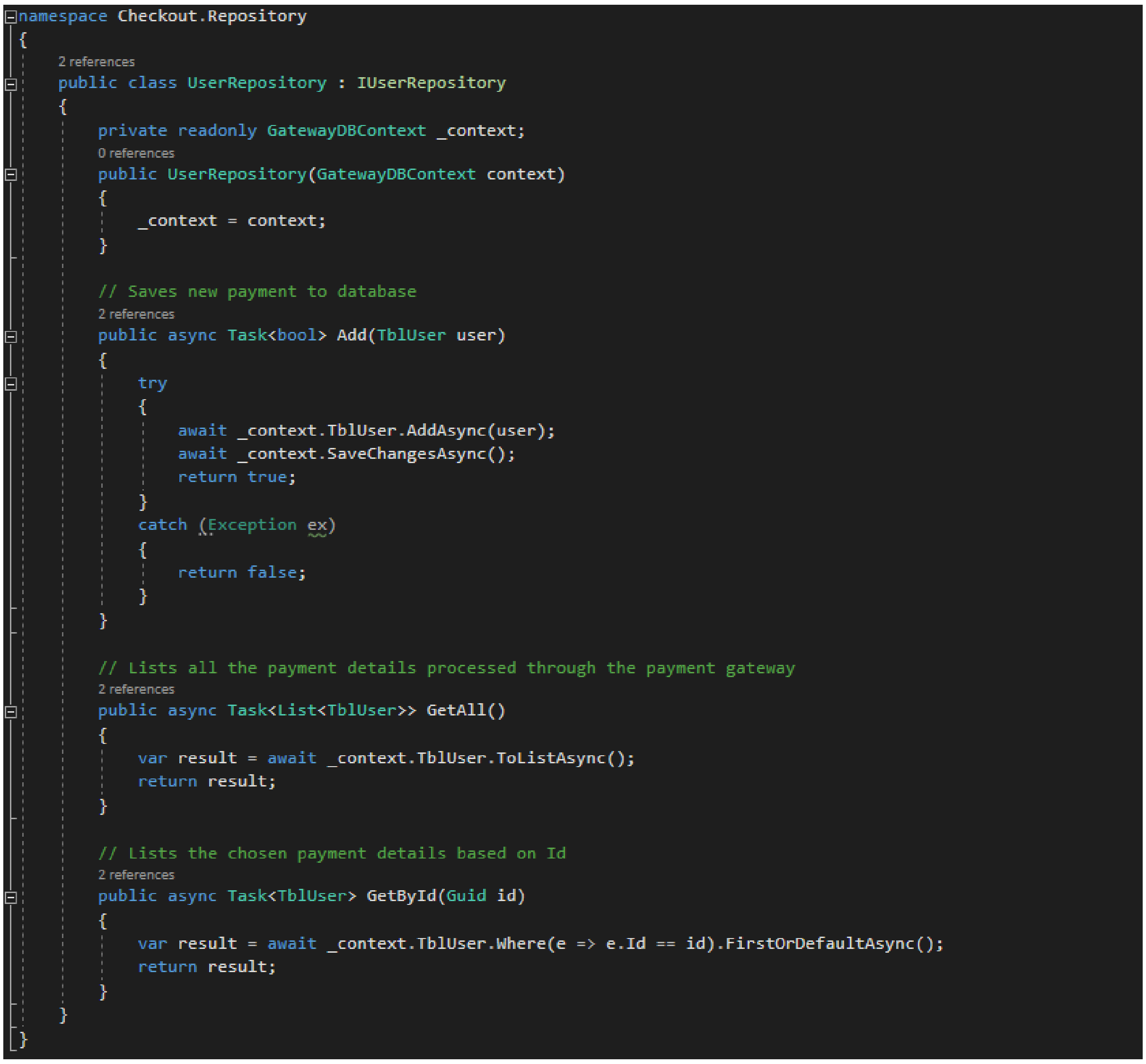
Thirdly, the class library “Checkout.Repository” will consist of the context class and database table model which is generated from the database created in Microsoft SQL Server Management Studio (SSMS). The database created in SSMS is shown in figure 4.



Column Name	Data Type	Allow Nulls
Id	uniqueidentifier	☐
FirstName	varchar(50)	☐
LastName	varchar(50)	☐
PhoneNumber	varchar(50)	☐
Address	varchar(50)	☐
Country	varchar(50)	☐
City	varchar(50)	☐
ZipCode	varchar(50)	☐
CardNumber	varchar(50)	☐
ExpiryDate	varchar(50)	☐
Cvv	varchar(50)	☐
Currency	varchar(50)	☐
Amount	decimal(18, 2)	☐

Figure 4. Application Database

The Checkout.Repository class library also consists of queries to retrieve and save data to the database shown in figure 4. Figure 5 shows code snippet to process a payment through the payment gateway and saves it to the database, and to retrieve details of previously made payments from the database.

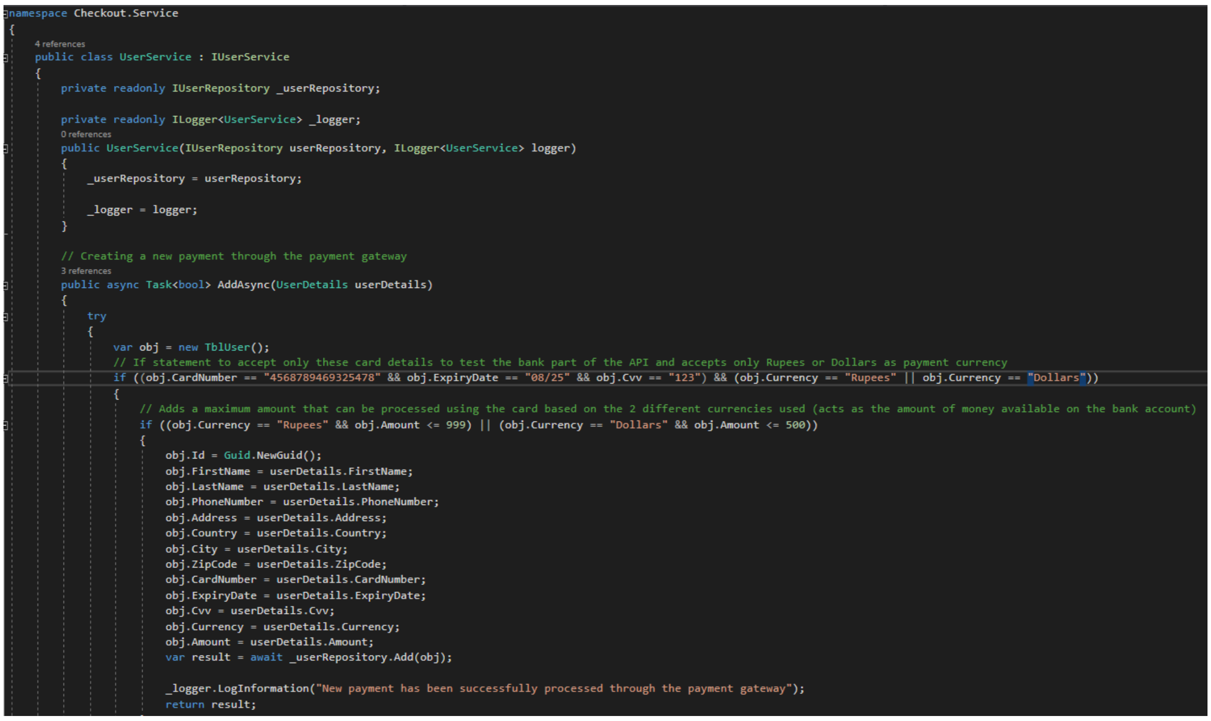


```
namespace Checkout.Repository
{
    2 references
    public class UserRepository : IUserRepository
    {
        private readonly GatewayDBContext _context;
        0 references
        public UserRepository(GatewayDBContext context)
        {
            _context = context;
        }

        // Saves new payment to database
        2 references
        public async Task<bool> Add(TblUser user)
        {
            try
            {
                await _context.TblUser.AddAsync(user);
                await _context.SaveChangesAsync();
                return true;
            }
            catch (Exception ex)
            {
                return false;
            }
        }

        // Lists all the payment details processed through the payment gateway
        2 references
        public async Task<List<TblUser>> GetAll()
        {
            var result = await _context.TblUser.ToListAsync();
            return result;
        }

        // Lists the chosen payment details based on Id
        2 references
        public async Task<TblUser> GetById(Guid id)
        {
            var result = await _context.TblUser.Where(e => e.Id == id).FirstOrDefaultAsync();
            return result;
        }
    }
}
```

Figure 5. Code Snippet to Add and Retrieve Payment

Checkout.Service Layer

The service layer of the application holds the operations to add and retrieve data which is used for communication between the UI layer and the Repository layer. Code snippet from the service layer is shown in figure 5, which holds the logic of the application.

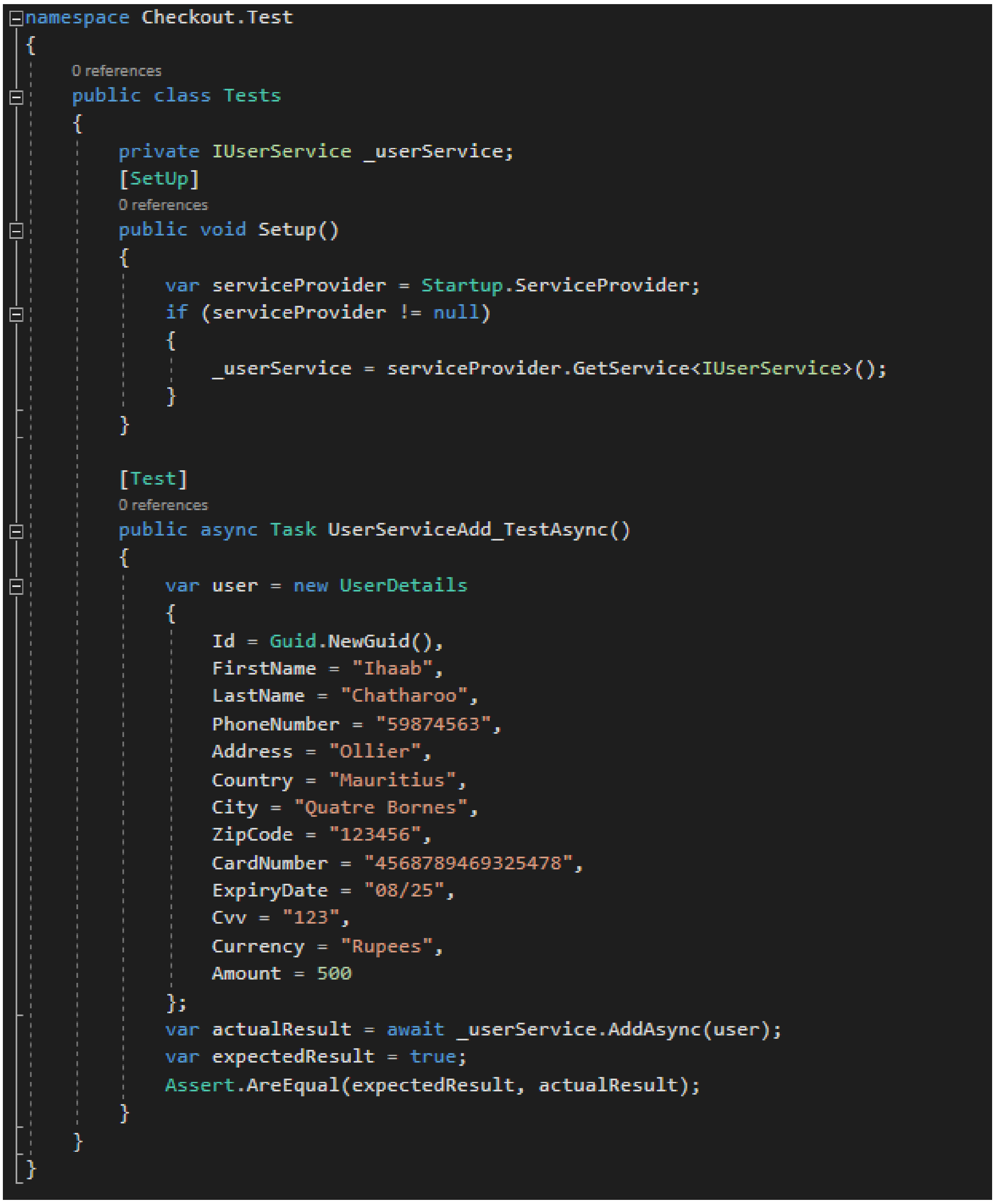


```
namespace Checkout.Service
{
    4 references
    public class UserService : IUserService
    {
        private readonly IUserRepository _userRepository;

        private readonly ILogger<UserService> _logger;
        0 references
        public UserService(IUserRepository userRepository, ILogger<UserService> logger)
        {
            _userRepository = userRepository;

            _logger = logger;
        }

        // Creating a new payment through the payment gateway
        3 references
        public async Task<bool> AddAsync(UserDetails userDetails)
        {
            try
            {
                var obj = new TblUser();
                // If statement to accept only these card details to test the bank part of the API and accepts only Rupees or Dollars as payment currency
                if ((obj.CardNumber == "4568789469325478" && obj.ExpiryDate == "08/25" && obj.Cvv == "123") && (obj.Currency == "Rupees" || obj.Currency == "Dollars"))
                {
                    // Adds a maximum amount that can be processed using the card based on the 2 different currencies used (acts as the amount of money available on the bank account)
                    if ((obj.Currency == "Rupees" && obj.Amount <= 999) || (obj.Currency == "Dollars" && obj.Amount <= 500))
                    {
                        obj.Id = Guid.NewGuid();
                        obj.FirstName = userDetails.FirstName;
                        obj.LastName = userDetails.LastName;
                        obj.PhoneNumber = userDetails.PhoneNumber;
                        obj.Address = userDetails.Address;
                        obj.Country = userDetails.Country;
                        obj.City = userDetails.City;
                        obj.ZipCode = userDetails.ZipCode;
                        obj.CardNumber = userDetails.CardNumber;
                        obj.ExpiryDate = userDetails.ExpiryDate;
                        obj.Cvv = userDetails.Cvv;
                        obj.Currency = userDetails.Currency;
                        obj.Amount = userDetails.Amount;
                        var result = await _userRepository.Add(obj);

                        _logger.LogInformation("New payment has been successfully processed through the payment gateway");
                        return result;
```

Figure 6. Code Snippet from the Service Layer

Checkout.Test Layer

This layer contains the unit test implemented for the application. The test done is to check whether the application successfully processes a payment, a successful response was obtained. Figure 6 shows the unit test implemented.

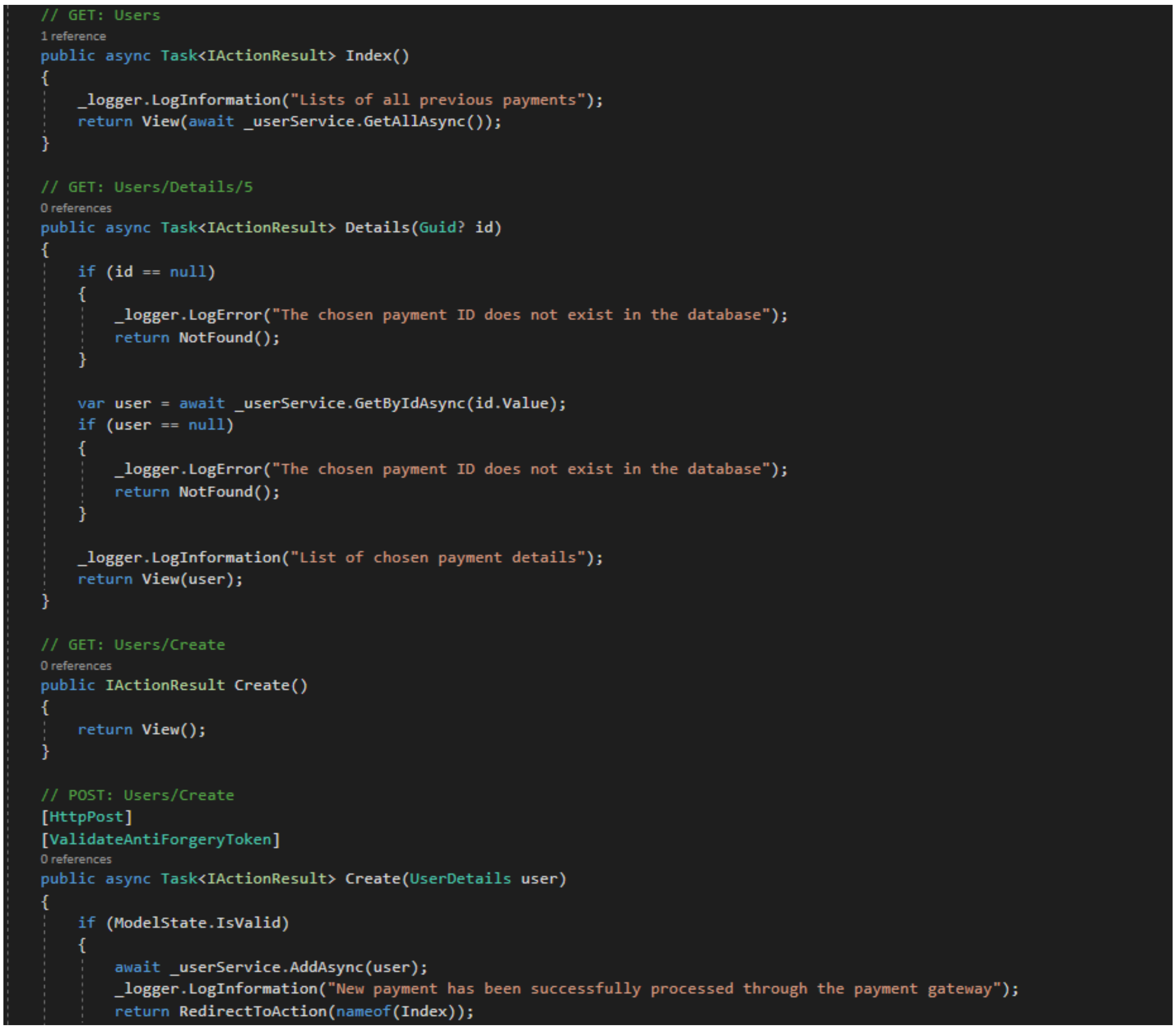


```
namespace Checkout.Test
{
    0 references
    public class Tests
    {
        private IUserService _userService;
        [SetUp]
        0 references
        public void Setup()
        {
            var serviceProvider = Startup.ServiceProvider;
            if (serviceProvider != null)
            {
                _userService = serviceProvider.GetService<IUserService>();
            }
        }

        [Test]
        0 references
        public async Task UserServiceAdd_TestAsync()
        {
            var user = new UserDetails
            {
                Id = Guid.NewGuid(),
                FirstName = "Ihaab",
                LastName = "Chatharoo",
                PhoneNumber = "59874563",
                Address = "Ollier",
                Country = "Mauritius",
                City = "Quatre Bornes",
                ZipCode = "123456",
                CardNumber = "4568789469325478",
                ExpiryDate = "08/25",
                Cvv = "123",
                Currency = "Rupees",
                Amount = 500
            };
            var actualResult = await _userService.AddAsync(user);
            var expectedResult = true;
            Assert.AreEqual(expectedResult, actualResult);
        }
    }
}
```

Figure 7. Unit Test Code Snippet

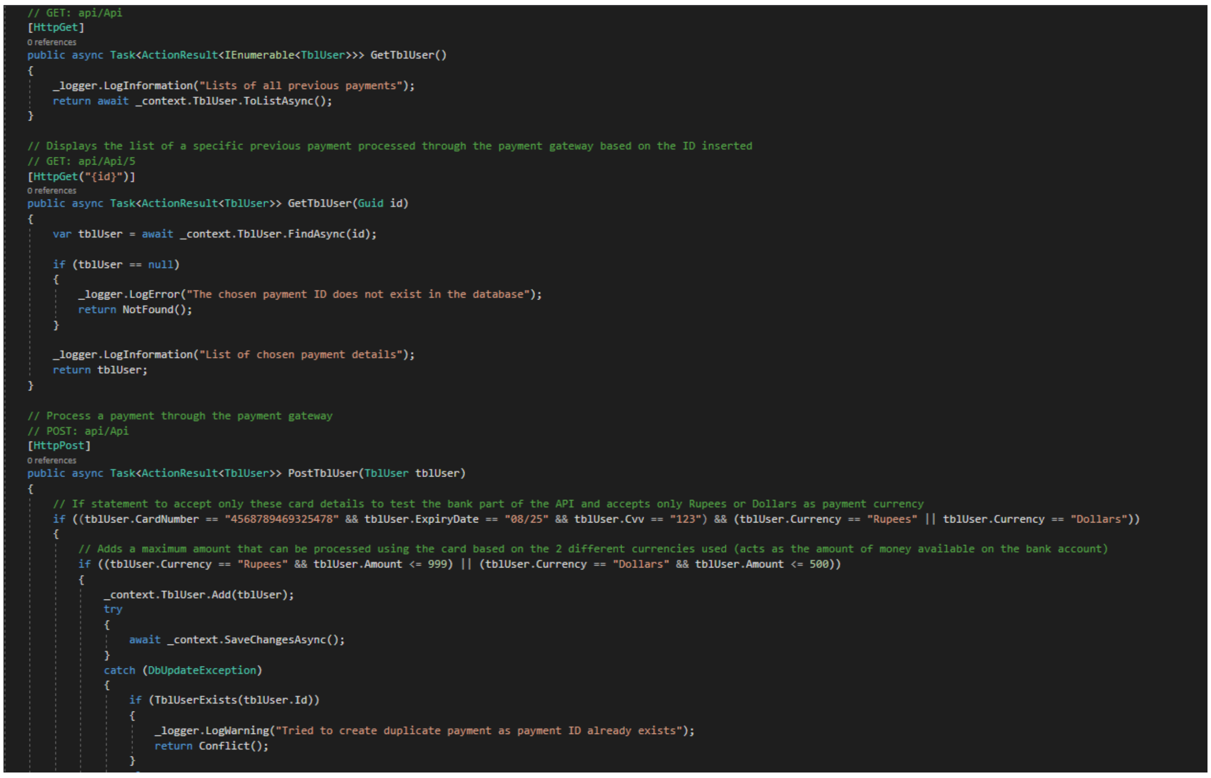
Checkout.Web Layer

This layer contains the web application to be used as the client-side for the payment gateway and the web API which is used with Postman as API client to get and post data to the database. Code snippets for the web application controller “UsersController” and the web API controller “ApiController” are shown in figures 8 and 9, respectively.



```
// GET: Users
1 reference
public async Task<IActionResult> Index()
{
    _logger.LogInformation("Lists of all previous payments");
    return View(await _userService.GetAllAsync());
}

// GET: Users/Details/5
0 references
public async Task<IActionResult> Details(Guid? id)
{
    if (id == null)
    {
        _logger.LogError("The chosen payment ID does not exist in the database");
        return NotFound();
    }

    var user = await _userService.GetByIdAsync(id.Value);
    if (user == null)
    {
        _logger.LogError("The chosen payment ID does not exist in the database");
        return NotFound();
    }

    _logger.LogInformation("List of chosen payment details");
    return View(user);
}

// GET: Users/Create
0 references
public IActionResult Create()
{
    return View();
}

// POST: Users/Create
[HttpPost]
[ValidateAntiForgeryToken]
0 references
public async Task<IActionResult> Create(UserDetails user)
{
    if (ModelState.IsValid)
    {
        await _userService.AddAsync(user);
        _logger.LogInformation("New payment has been successfully processed through the payment gateway");
        return RedirectToAction(nameof(Index));
```

Figure 8. UsersController Code Snippet



```csharp
// GET: api/Api
[HttpGet]
0 references
public async Task<ActionResult<IEnumerable<TblUser>>> GetTblUser()
{
    _logger.LogInformation("Lists of all previous payments");
    return await _context.TblUser.ToListAsync();
}

// Displays the list of a specific previous payment processed through the payment gateway based on the ID inserted
// GET: api/Api/5
[HttpGet("{id}")]
0 references
public async Task<ActionResult<TblUser>> GetTblUser(Guid id)
{
    var tblUser = await _context.TblUser.FindAsync(id);

    if (tblUser == null)
    {
        _logger.LogError("The chosen payment ID does not exist in the database");
        return NotFound();
    }

    _logger.LogInformation("List of chosen payment details");
    return tblUser;
}

// Process a payment through the payment gateway
// POST: api/Api
[HttpPost]
0 references
public async Task<ActionResult<TblUser>> PostTblUser(TblUser tblUser)
{
    // If statement to accept only these card details to test the bank part of the API and accepts only Rupees or Dollars as payment currency
    if ((tblUser.CardNumber == "4568789469325478" && tblUser.ExpiryDate == "08/25" && tblUser.Cvv == "123") && (tblUser.Currency == "Rupees" || tblUser.Currency == "Dollars"))
    {
        // Adds a maximum amount that can be processed using the card based on the 2 different currencies used (acts as the amount of money available on the bank account)
        if ((tblUser.Currency == "Rupees" && tblUser.Amount <= 999) || (tblUser.Currency == "Dollars" && tblUser.Amount <= 500))
        {
            _context.TblUser.Add(tblUser);
            try
            {
                await _context.SaveChangesAsync();
            }
            catch (DbUpdateException)
            {
                if (TblUserExists(tblUser.Id))
                {
                    _logger.LogWarning("Tried to create duplicate payment as payment ID already exists");
                    return Conflict();
                }
```

Figure 9. ApiController Code Snippet

Application Functionalities

Bank Simulator

The bank part of the application is hardcoded using if-statement so that the proper responses are obtained to be able to fully test the API. The values used for testing of the API are:

- Card number : 4568789469325478
- Card expiry date : 08/25
- Cvv : 123
- Currency : Rupees or Dollars

Furthermore, when processing a payment if the amount is in rupees and it is more than 999, it will return an error saying that there are insufficient funds in the account. If the amount paid is in dollars, the maximum value is 500.

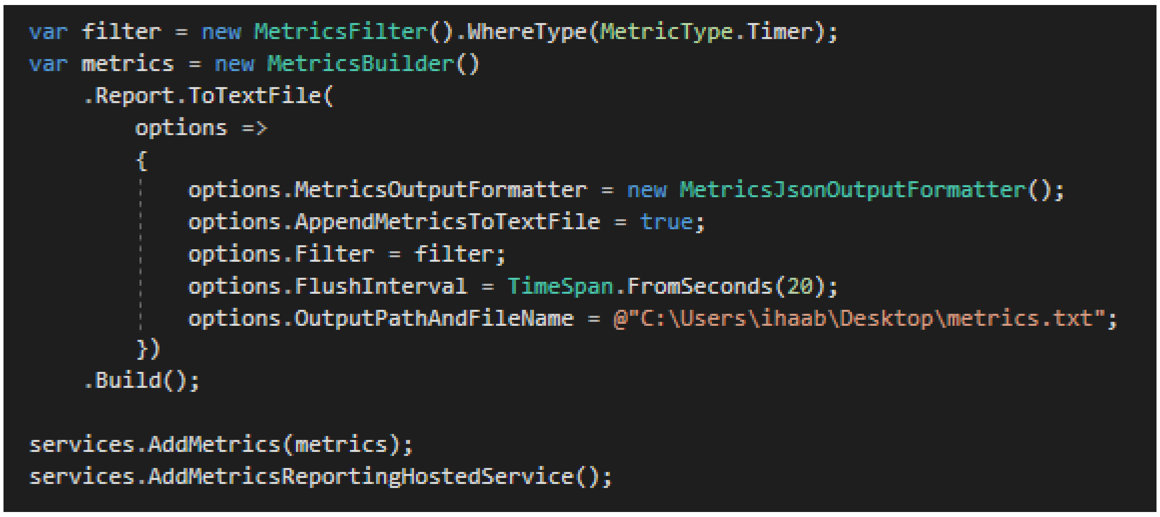
However, this could be improved by creating another table in the database of the application for the card details of several users. In this manner, when a purchase has been processed the number can be reduced from the database, thus a more realistic approach for the bank simulator.

Application Logging

Application logs have been implemented using Microsoft logging package. The logs help to know what is going on in the application by writing logs to the console. Application logs also helps to make debugging easier.

Application Metrics

The implementation of application metrics helps to detect performance issues in the application as it records the time taken for the requests to be processed. The application metrics used saves the results in a text file where the results obtained can be easily read. Figure 10 shows the code for the application metrics implemented in the application.

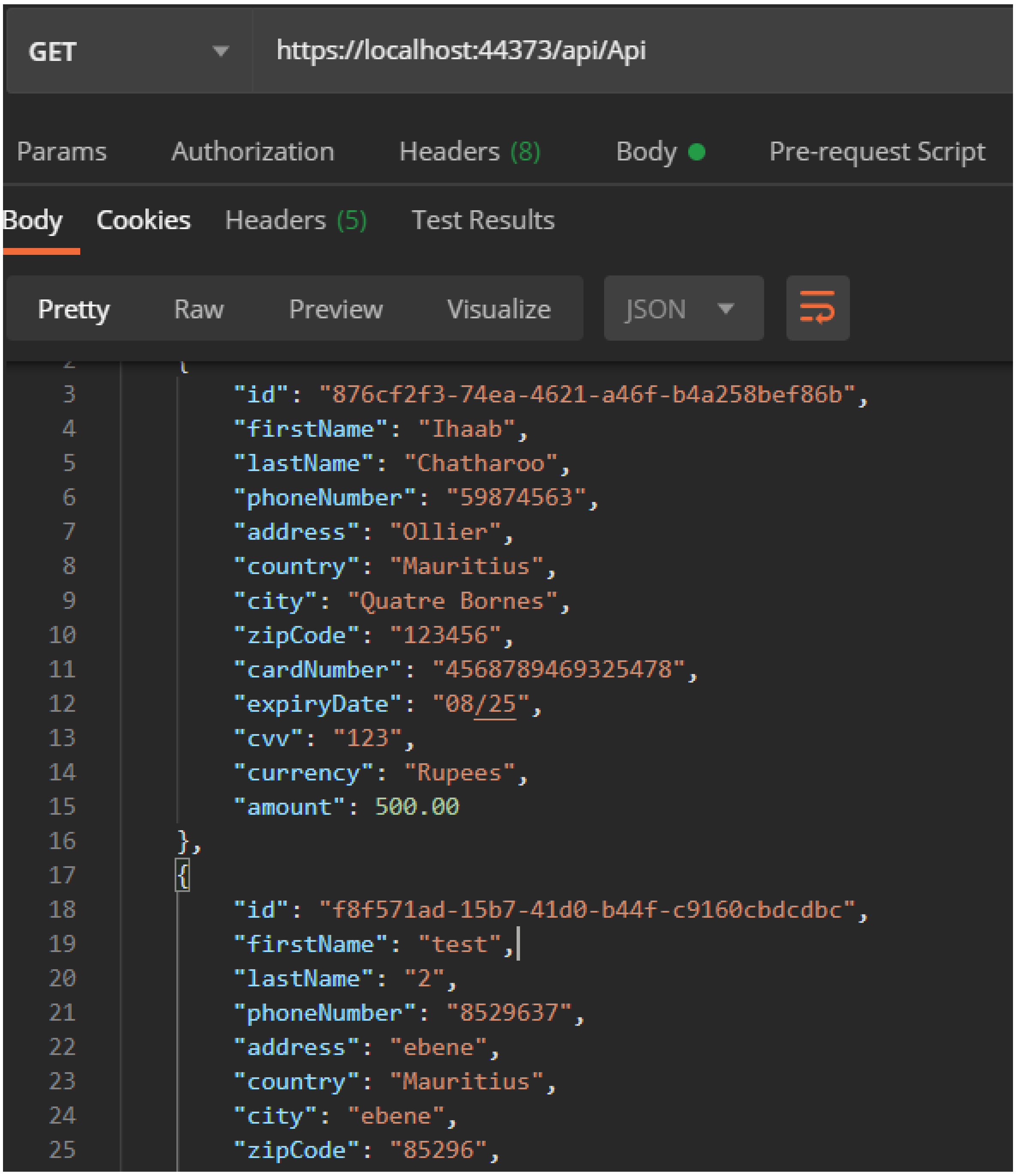


```
var filter = new MetricsFilter().WhereType(MetricType.Timer);
var metrics = new MetricsBuilder()
    .Report.ToTextFile(
        options =>
        {
            options.MetricsOutputFormatter = new MetricsJsonOutputFormatter();
            options.AppendMetricsToTextFile = true;
            options.Filter = filter;
            options.FlushInterval = TimeSpan.FromSeconds(20);
            options.OutputPathAndFileName = @"C:\Users\ihaab\Desktop\metrics.txt";
        })
    .Build();

services.AddMetrics(metrics);
services.AddMetricsReportingHostedService();
```

Figure 10. Application Metrics Code Snippet

Application Request Results

The API is tested using Postman as API client to perform the GET and POST requests. To perform the requests using Postman, the following URL “https://localhost:44373/api/Api” must be entered in Postman.



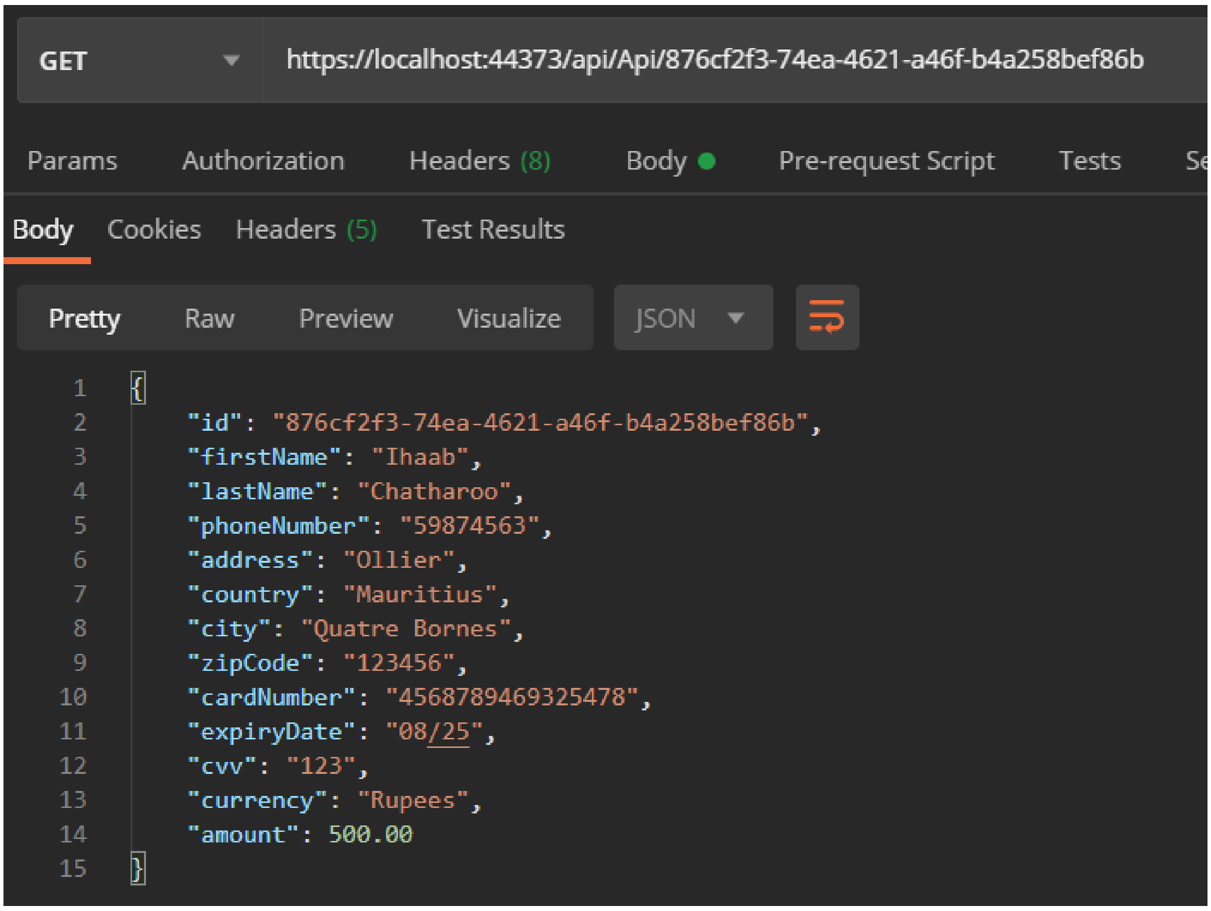
https://localhost:44373/api/Api

```
        "id": "876cf2f3-74ea-4621-a46f-b4a258bef86b",
        "firstName": "Ihaab",
        "lastName": "Chatharoo",
        "phoneNumber": "59874563",
        "address": "Ollier",
        "country": "Mauritius",
        "city": "Quatre Bornes",
        "zipCode": "123456",
        "cardNumber": "4568789469325478",
        "expiryDate": "08/25",
        "cvv": "123",
        "currency": "Rupees",
        "amount": 500.00
    },
    {
        "id": "f8f571ad-15b7-41d0-b44f-c9160cbdcdbc",
        "firstName": "test",
        "lastName": "2",
        "phoneNumber": "8529637",
        "address": "ebene",
        "country": "Mauritius",
        "city": "ebene",
        "zipCode": "85296",
```

Figure 11. GET Request in Postman

Figure 11 shows the GET result obtained when run in Postman. This request displays all the previous payment made through the payment gateway application.

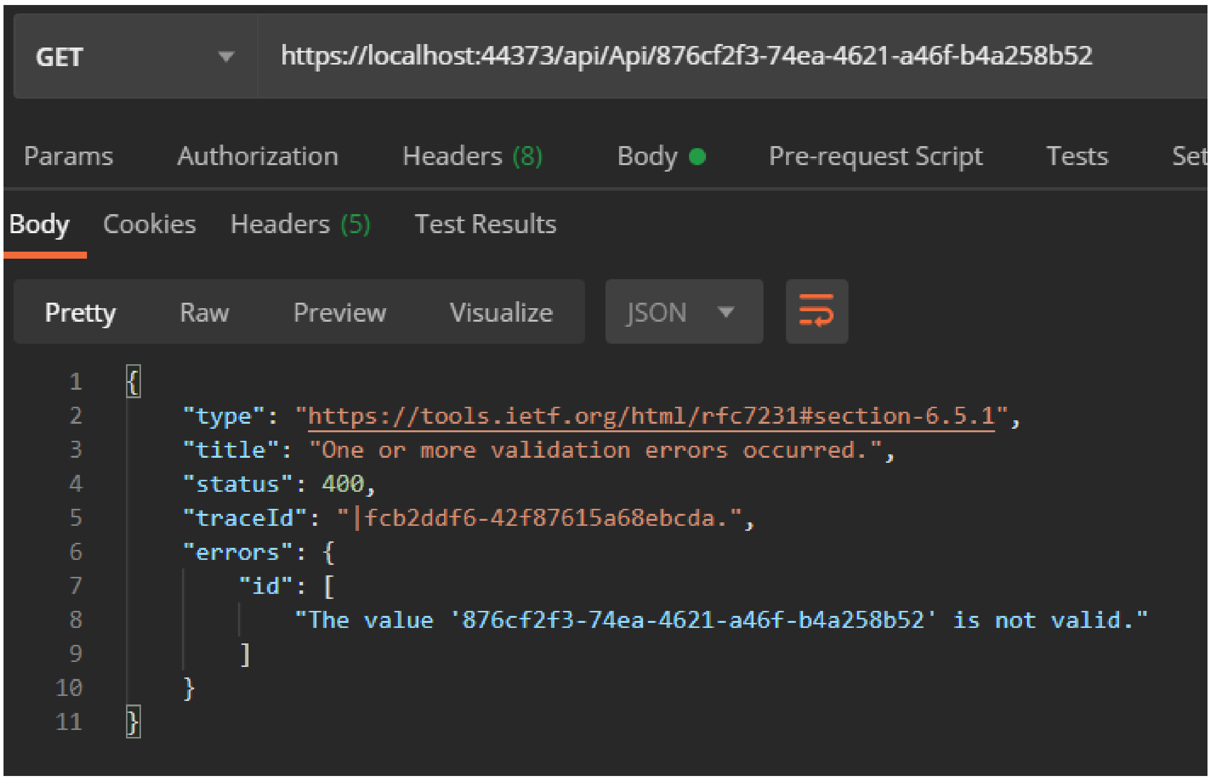
The next request performed is to get the details of a previously made payment based on its identifier which is shown in figure 12. Figure 13 shows the result obtained when an unknown identifier is entered in the URL field, the API client returns a bad request and specifies what the error was.



```
https://localhost:44373/api/Api/876cf2f3-74ea-4621-a46f-b4a258bef86b
```

```
{
    "id": "876cf2f3-74ea-4621-a46f-b4a258bef86b",
    "firstName": "Ihaab",
    "lastName": "Chatharoo",
    "phoneNumber": "59874563",
    "address": "Ollier",
    "country": "Mauritius",
    "city": "Quatre Bornes",
    "zipCode": "123456",
    "cardNumber": "4568789469325478",
    "expiryDate": "08/25",
    "cvv": "123",
    "currency": "Rupees",
    "amount": 500.00
}
```

Figure 12. Get Payment from Identifier

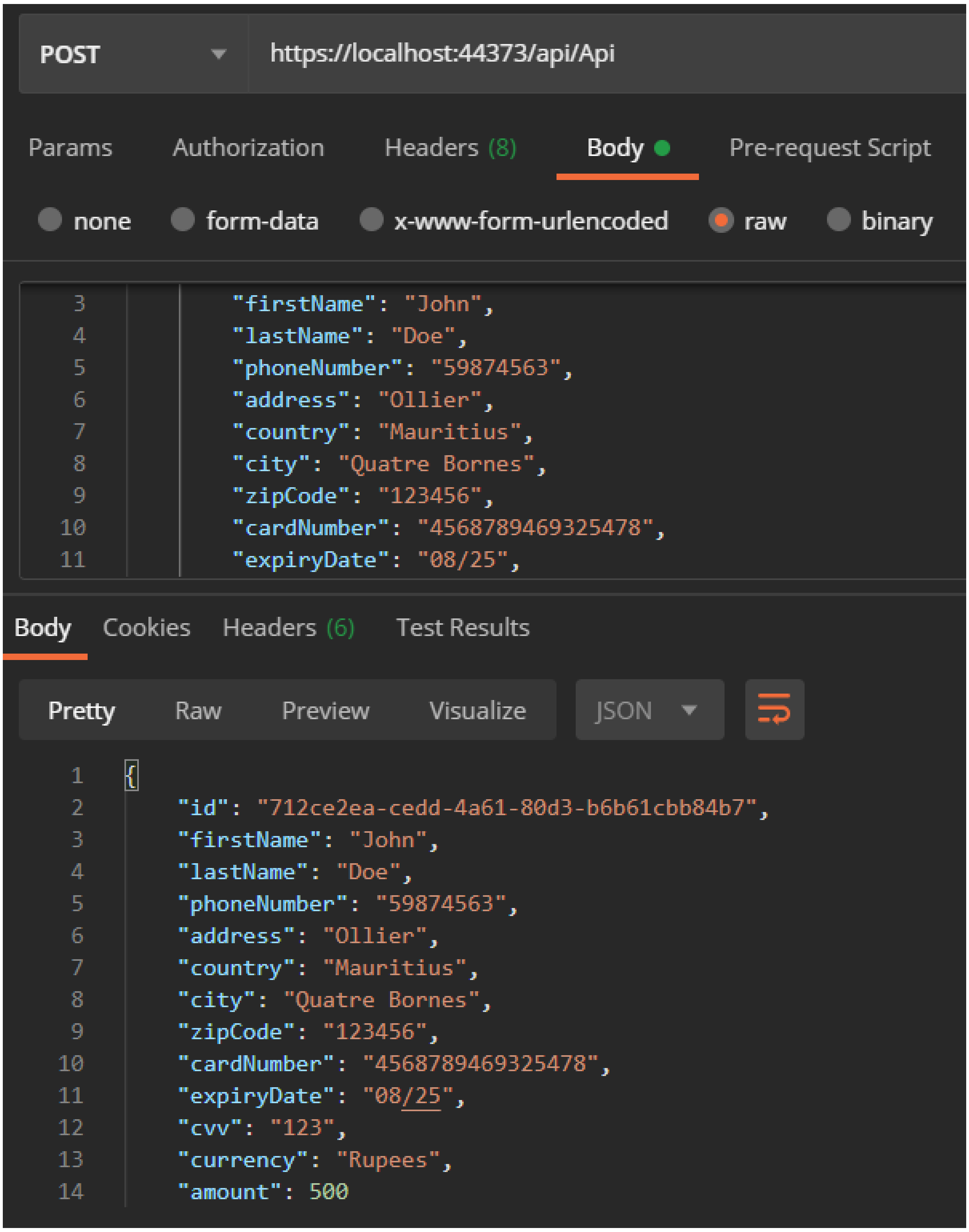


```
https://localhost:44373/api/Api/876cf2f3-74ea-4621-a46f-b4a258b52
```

```
{
    "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
    "title": "One or more validation errors occurred.",
    "status": 400,
    "traceId": "|fcb2ddf6-42f87615a68ebcda.",
    "errors": {
        "id": [
            "The value '876cf2f3-74ea-4621-a46f-b4a258b52' is not valid."
        ]
    }
}
```

Figure 13. Get Payment from Unknown Identifier

Finally, the last test to be performed on the application is to perform a POST request. Firstly, a POST request will be performed with all the correct values entered which is shown in figure 14.



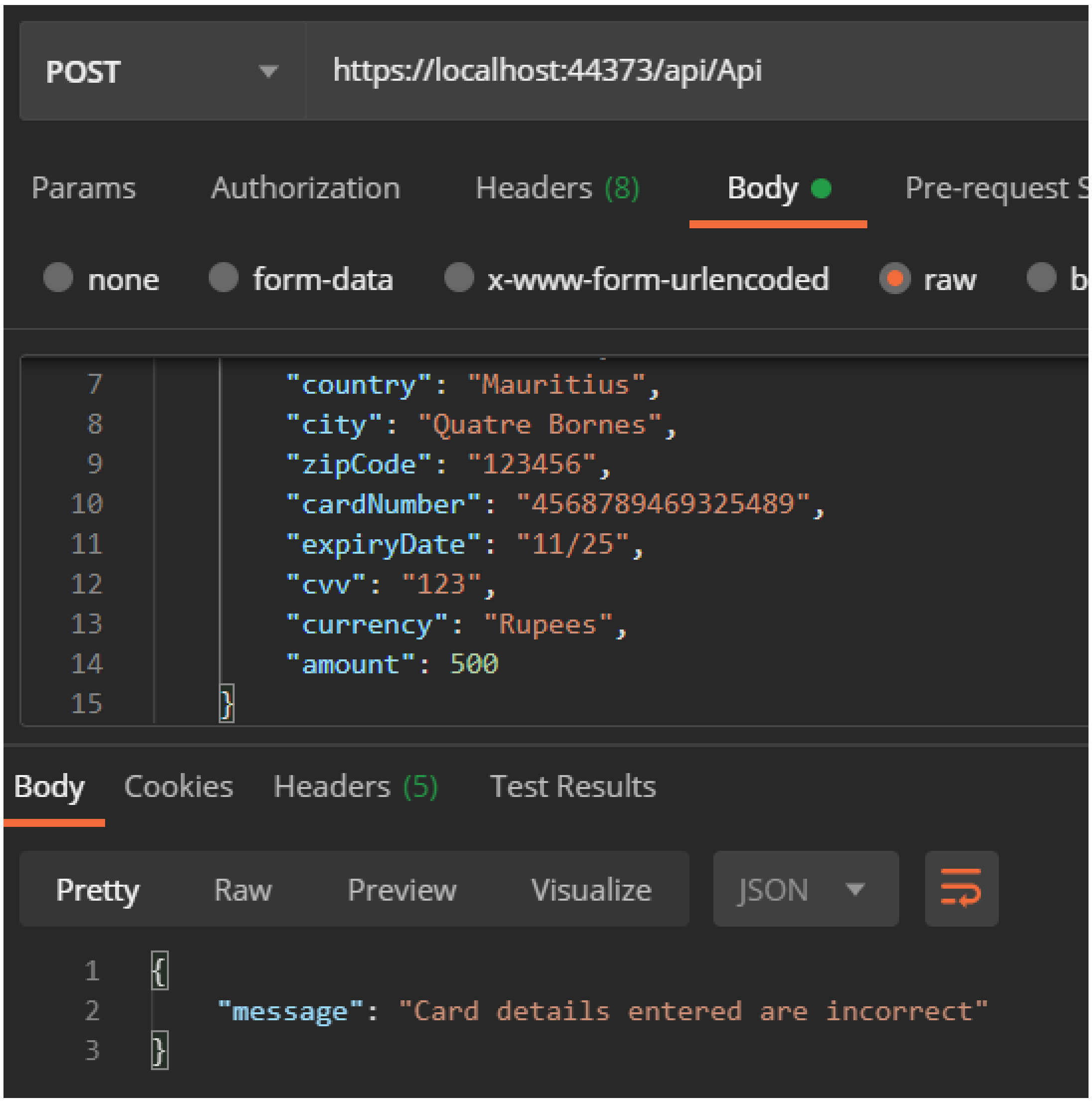
```
https://localhost:44373/api/Api
```

```
        "firstName": "John",
        "lastName": "Doe",
        "phoneNumber": "59874563",
        "address": "Ollier",
        "country": "Mauritius",
        "city": "Quatre Bornes",
        "zipCode": "123456",
        "cardNumber": "4568789469325478",
        "expiryDate": "08/25",
```

```
{
    "id": "712ce2ea-cedd-4a61-80d3-b6b61cbb84b7",
    "firstName": "John",
    "lastName": "Doe",
    "phoneNumber": "59874563",
    "address": "Ollier",
    "country": "Mauritius",
    "city": "Quatre Bornes",
    "zipCode": "123456",
    "cardNumber": "4568789469325478",
    "expiryDate": "08/25",
    "cvv": "123",
    "currency": "Rupees",
    "amount": 500
```

Figure 14. POST Request Result from Postman

The next test consists of entering incorrect data for the card details, this will return an error and a message stating what was wrong in the transaction, this is shown in figure 15.



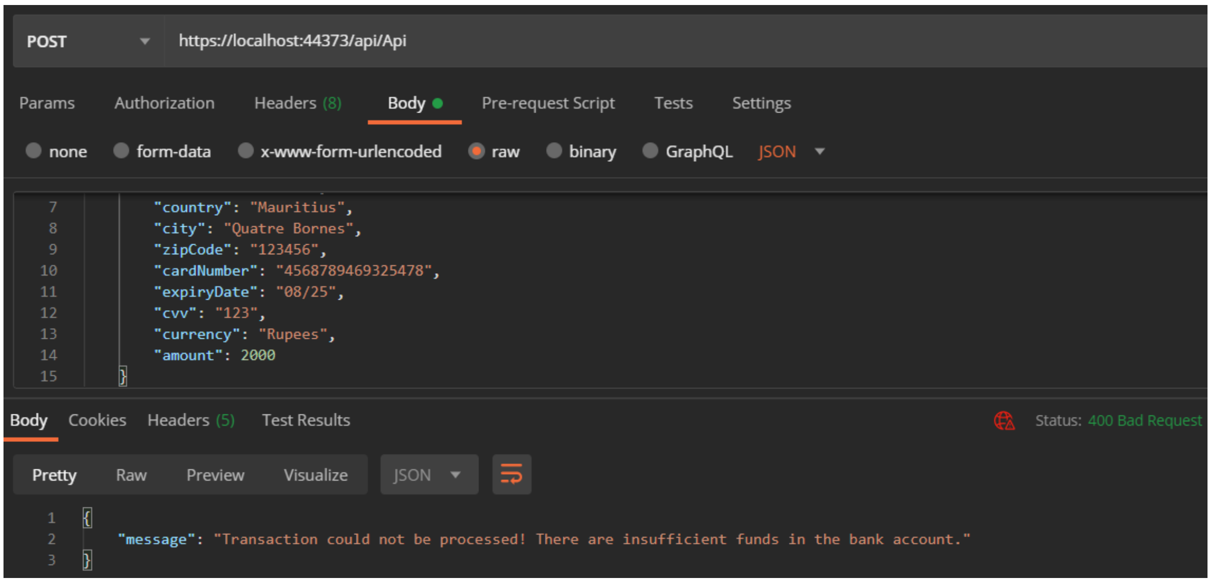
```
https://localhost:44373/api/Api
```

```
    "country": "Mauritius",
    "city": "Quatre Bornes",
    "zipCode": "123456",
    "cardNumber": "4568789469325489",
    "expiryDate": "11/25",
    "cvv": "123",
    "currency": "Rupees",
    "amount": 500
}
```

```
{
    "message": "Card details entered are incorrect"
}
```

Figure 15. POST Request Result with Invalid Card Details

Finally, the last POST request consists of entering an amount more than what is currently on the card, this will also return an error stating that there are not sufficient funds on the account, shown in figure 16.



https://localhost:44373/api/Api

```
    "country": "Mauritius",
    "city": "Quatre Bornes",
    "zipCode": "123456",
    "cardNumber": "4568789469325478",
    "expiryDate": "08/25",
    "cvv": "123",
    "currency": "Rupees",
    "amount": 2000
}
```

Status: 400 Bad Request

```
{
    "message": "Transaction could not be processed! There are insufficient funds in the bank account."
}
```

Figure 16. POST Request Result with Insufficient Funds in Bank Account