Building Payment Functionality with Spring Boot: Step-by-Step Guide

Umasree Kollu · Follow
5 min read · Apr 3, 2024

Online Payment

Welcome to 'Building Payment Functionality with Spring Boot: Step-by-Step Guide'! In this tutorial, we'll navigate through the intricate process of seamlessly integrating payment functionality into your Spring Boot applications. From setting up your environment to deploying a robust payment system, this guide offers a comprehensive roadmap for developers seeking to enhance their web applications with secure and efficient payment processing. Let's dive in and unlock the power of payment integration with Spring Boot!

Getting Started

Step 1: Defining the Payment Entity

First, we need an entity to represent payments in our application. This entity will map to a database table.

```
package com.payment.entity;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Payment {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String number;
    private String email;
    private String address;
    private int billValue;
    private String cardNumber;
    private String cardHolder;
    private String dateValue;
    private String cvc;
}
```

Explanation:

- We've annotated the class with **`@Entity`** to mark it as a JPA entity, meaning it will be mapped to a database table.
- **`@Id`** marks the `id` field as the primary key.
- **`@GeneratedValue(strategy = GenerationType.IDENTITY)`** specifies that the ID should

be automatically generated by the database.

- The other fields represent the attributes of a payment, such as **`name`**, **`number`**, **`email`**, etc.

Step 2: Creating the Payment Repository

Next, we need a repository interface to interact with the **`Payment`** entity and perform CRUD operations.

```
package com.vizen.repository;

import com.vizen.entity.Payment;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface PaymentRepository extends JpaRepository<Payment, Long> {
}
```

Explanation:

- We extend the **`JpaRepository`** interface, which provides methods for common database operations (CRUD).
- The first type parameter (**`Payment`**) specifies the entity type, and the second type parameter (**`Long`**) specifies the type of the primary key.

Step 3: Creating Request DTO (Data Transfer Object)

We need a DTO to represent the data sent by clients when creating a payment.

```
package com.vizen.request.dto;

import io.swagger.annotations.ApiModel;
import io.swagger.annotations.ApiModelProperty;
import jakarta.validation.constraints.*;
import lombok.Data;
```

```
@Data
@ApiModel(value = "PaymentRequestDTO", description = "Parameters required for a
public class PaymentCreateRequest {

    @ApiModelProperty(notes = "Name of the payment", required = true)
    @NotBlank(message = "Name is required")
    @Size(max = 255)
    private String name;

    @ApiModelProperty(notes = "Number of the payment")
    private String number;

    @ApiModelProperty(notes = "Email of the payment")
    private String email;

    @ApiModelProperty(notes = "Address of the payment")
    private String address;

    @ApiModelProperty(notes = "Bill value of the payment")
    private int billValue;

    @ApiModelProperty(notes = "Card number of the payment")
    private String cardNumber;

    @ApiModelProperty(notes = "Card holder of the payment")
    private String cardHolder;

    @ApiModelProperty(notes = "Date value of the payment")
    private String dateValue;

    @ApiModelProperty(notes = "CVC of the payment")
    private String cvc;
}
```

Explanation:

- This class represents a Data Transfer Object (DTO) used for transferring payment-related data between the client and the server.
- The `@Data` annotation from the Lombok library automatically generates getters, setters, and other boilerplate code.
- The `@ApiModel` annotation defines the DTO as a model for Swagger documentation, providing a name and description.

- Each field in the DTO is annotated with `@ApiModelProperty`, providing additional information about the field such as notes and whether it's required.
- Validation annotations such as `@NotBlank` and `@Size` are used to enforce constraints on the data. For example, the `name` field must not be blank and must not exceed 255 characters in length.
- Other fields such as `number`, `email`, `address`, `billValue`, `cardNumber`, `cardHolder`, `dateValue`, and `cvc` represent various attributes of a payment and are self-explanatory.

Step 4: Creating the Payment Service Interface and Implementation

We need a service interface and its implementation to handle business logic related to payments.

PaymentService.java

```
package com.vizen.service;

import com.vizen.request.dto.PaymentCreateRequest;
import com.vizen.response.dto.PaymentResponseDTO;

public interface PaymentService {
    PaymentResponseDTO createPayment(PaymentCreateRequest paymentRequestDTO);
}
```

PaymentServiceImpl.java

```
package com.vizen.service;

import com.vizen.entity.Payment;
import com.vizen.repository.PaymentRepository;
import com.vizen.request.dto.PaymentCreateRequest;
import com.vizen.response.dto.PaymentResponseDTO;
import org.springframework.beans.BeanUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
```

```
@Service
public class PaymentServiceImpl implements PaymentService {

    private final PaymentRepository paymentRepository;

    @Autowired
    public PaymentServiceImpl(PaymentRepository paymentRepository) {
        this.paymentRepository = paymentRepository;
    }

    @Override
    public PaymentResponseDTO createPayment(PaymentCreateRequest paymentRequest
        Payment payment = convertToEntity(paymentRequestDTO);
        Payment savedPayment = paymentRepository.save(payment);
        return convertToResponseDTO(savedPayment);
    }

    private Payment convertToEntity(PaymentCreateRequest paymentRequestDTO) {
        Payment payment = new Payment();
        BeanUtils.copyProperties(paymentRequestDTO, payment);
        return payment;
    }

    private PaymentResponseDTO convertToResponseDTO(Payment payment) {
        PaymentResponseDTO responseDTO = new PaymentResponseDTO();
        BeanUtils.copyProperties(payment, responseDTO);
        return responseDTO;
    }
}
```

Explanation:

- The **`PaymentService`** interface defines a method **`createPayment`** for creating payments.
- The **`PaymentServiceImpl`** class implements the **`PaymentService`** interface and provides the implementation for the **`createPayment`** method.
- In the **`createPayment`** method, we convert the **`PaymentCreateRequest`** DTO to a **`Payment`** entity, save it to the database using the repository, and then convert the saved entity to a response DTO before returning it.

Step 6: Creating the Payment Controller

Next create the **PaymentController** which handles HTTP requests related to payment operations.

```
package com.vizen.controller;

import com.vizen.request.dto.PaymentCreateRequest;
import com.vizen.response.dto.PaymentResponseDTO;
import com.vizen.service.PaymentService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/payments")
public class PaymentController {

    private final PaymentService paymentService;

    @Autowired
    public PaymentController(PaymentService paymentService) {
        this.paymentService = paymentService;
    }

    @PostMapping("/create")
    public ResponseEntity<PaymentResponseDTO> createPayment(@RequestBody Paymen
        PaymentResponseDTO createdPayment = paymentService.createPayment(paymen
        return new ResponseEntity<>(createdPayment, HttpStatus.CREATED);
    }
}
```

Explanation:

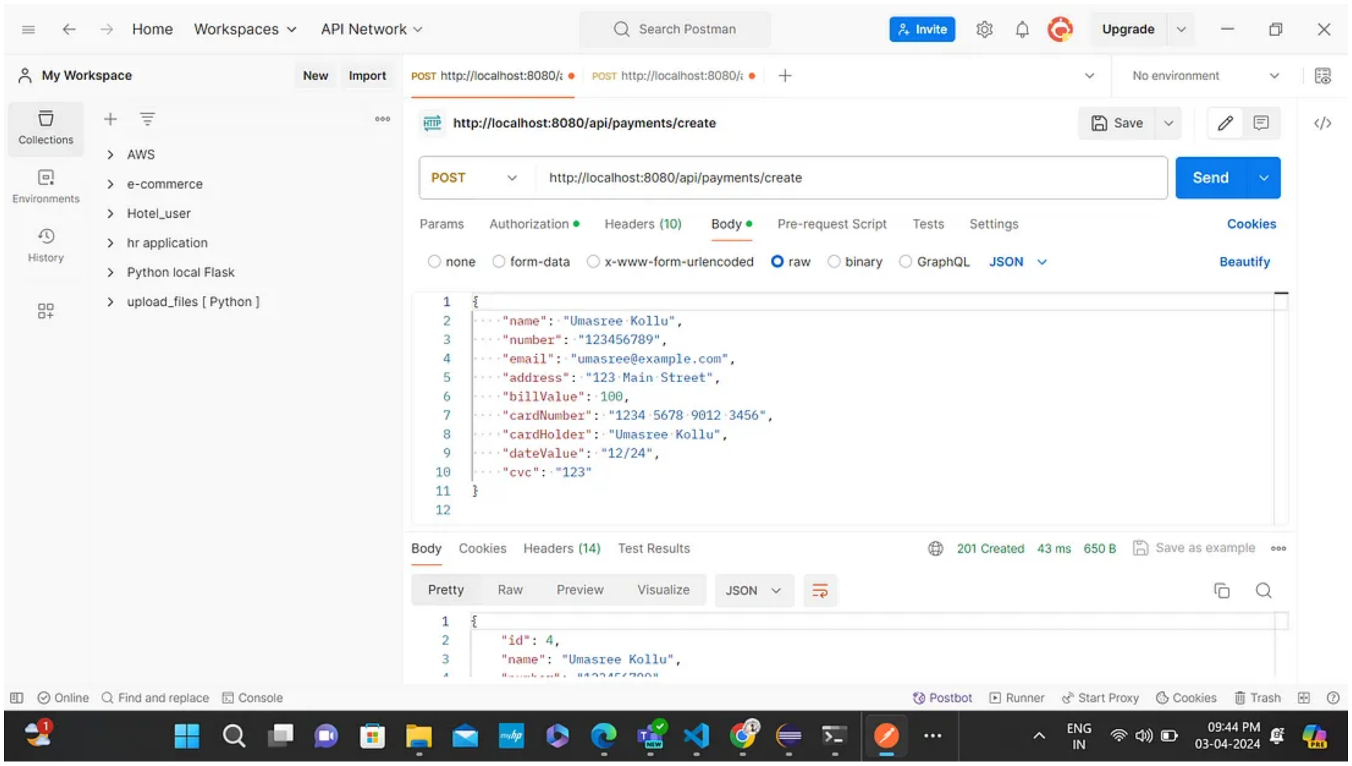
- Annotating the class with **`@RestController`** indicates that it's a controller component that handles HTTP requests and returns responses.
- **`@RequestMapping("/api/payments")`** sets the base URI for the controller's endpoints.

- `@Autowired` annotation injects the **PaymentService** dependency into the controller.
- The `createPayment` method handles POST requests to the `/api/payments/create` endpoint.
- `@RequestBody` annotation binds the request body to the `PaymentCreateRequest` object.
- The `createPayment` method delegates the creation of the payment to the `PaymentService` and returns the created payment as a response with HTTP status code 201 (CREATED).

This controller allows clients to create payments by sending a POST request with payment details in the request body. The payment creation logic is delegated to the `PaymentService`, keeping the controller lightweight and focused on handling HTTP requests.

Step 7: Testing the Payment Endpoint in Postman

Use Postman to send a POST request with payment details to your payment creation endpoint and verify the response to ensure payments are created successfully.



POST http://localhost:8080/api/payments/create

Body: ○ none ○ form-data ○ x-www-form-urlencoded ● raw ○ binary ○ GraphQL JSON

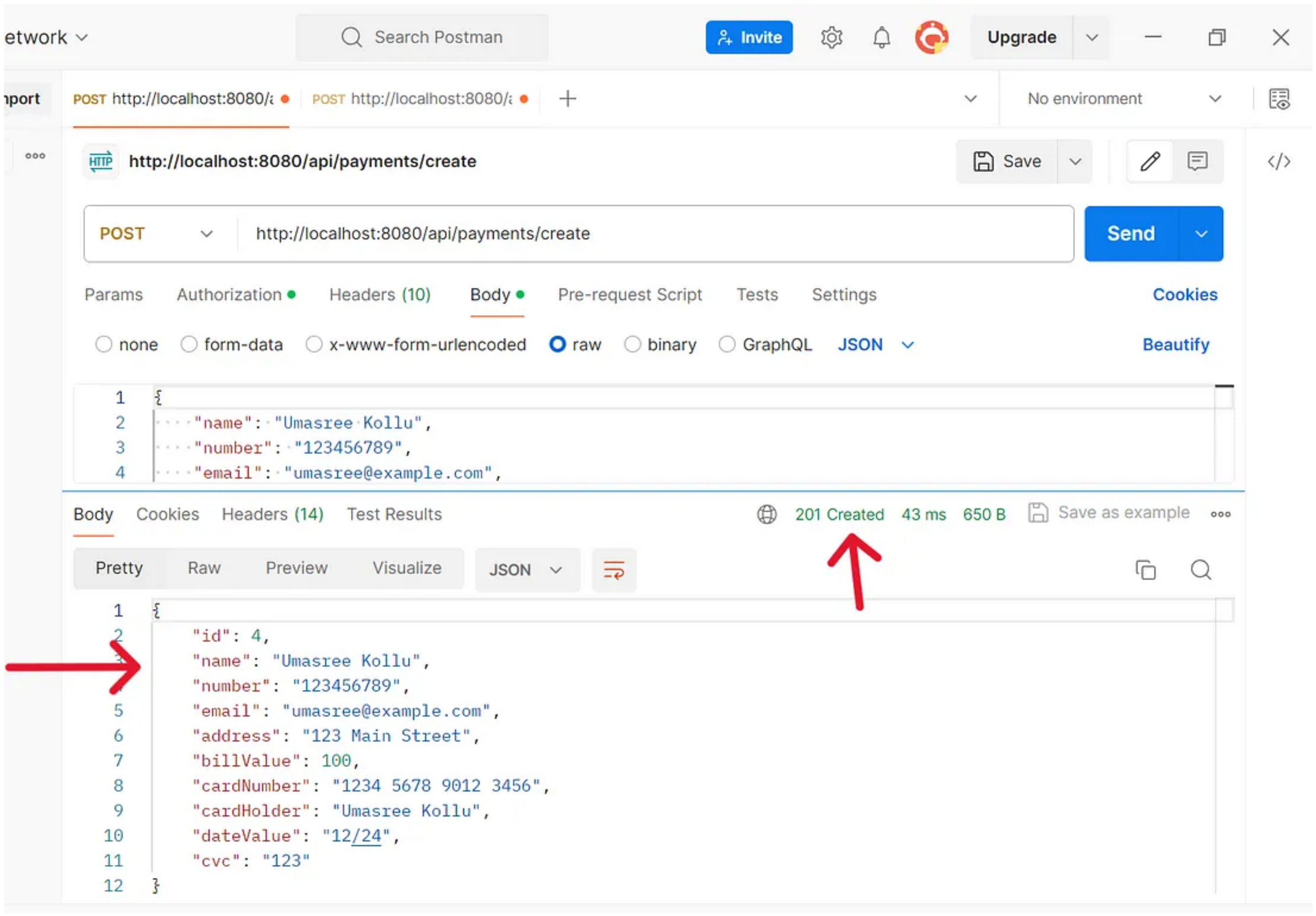
```
{
    "name": "Umasree Kollu",
    "number": "123456789",
    "email": "umasree@example.com",
    "address": "123 Main Street",
    "billValue": 100,
    "cardNumber": "1234 5678 9012 3456",
    "cardHolder": "Umasree Kollu",
    "dateValue": "12/24",
    "cvc": "123"
}
```

Body Cookies Headers (14) Test Results — 201 Created 43 ms 650 B

```
{
    "id": 4,
    "name": "Umasree Kollu",
```

Create payment using request JSON Data

Response data



http://localhost:8080/api/payments/create

POST http://localhost:8080/api/payments/create

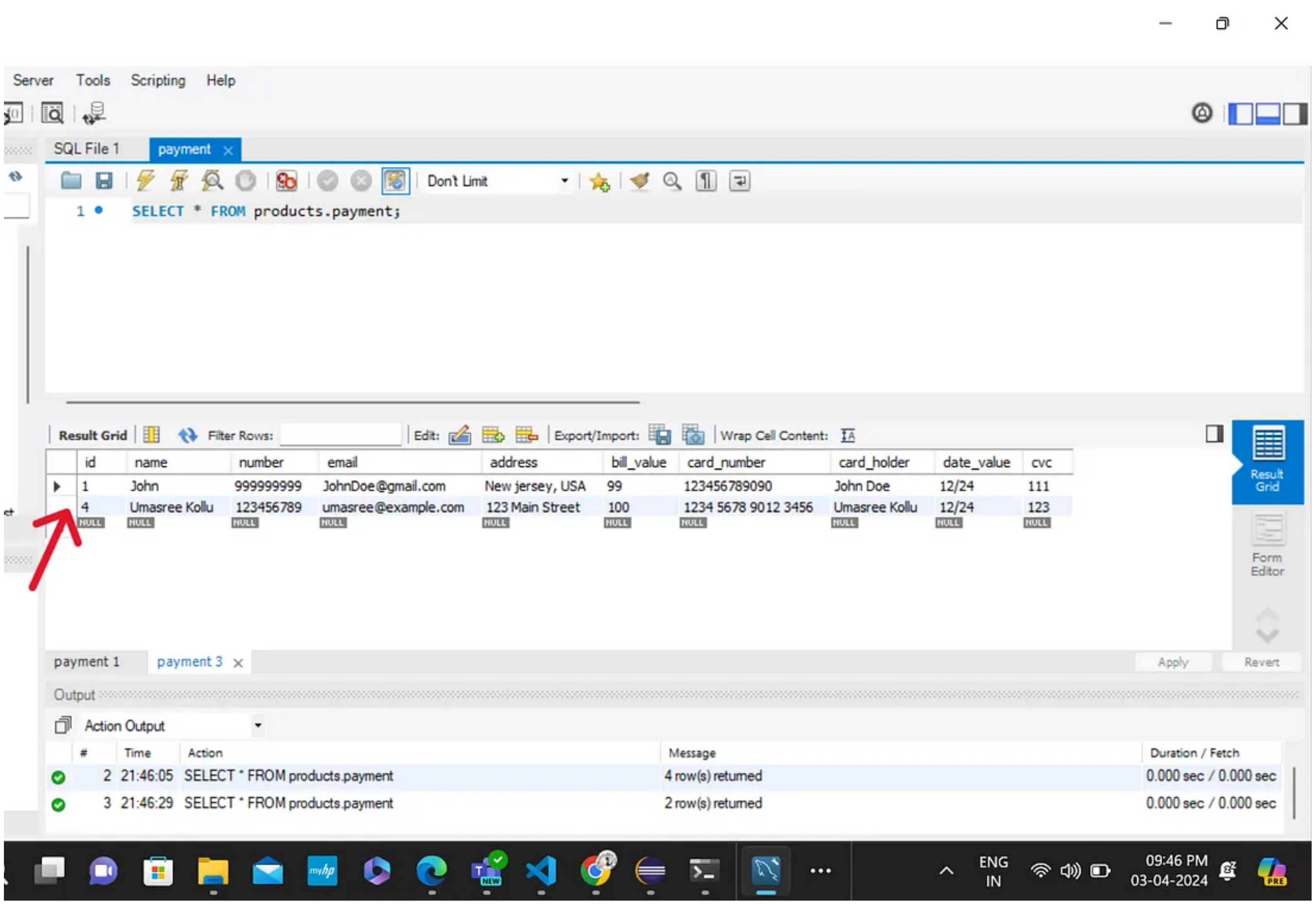
```
{
    "name": "Umasree Kollu",
    "number": "123456789",
    "email": "umasree@example.com",
```

201 Created 43 ms 650 B

```
{
    "id": 4,
    "name": "Umasree Kollu",
    "number": "123456789",
    "email": "umasree@example.com",
    "address": "123 Main Street",
    "billValue": 100,
    "cardNumber": "1234 5678 9012 3456",
    "cardHolder": "Umasree Kollu",
    "dateValue": "12/24",
    "cvc": "123"
}
```

Response data

After receiving the payment details, the Spring Boot application stores the information in a MySQL database, ensuring persistence and accessibility for future use.



```
SELECT * FROM products.payment;
```

id	name	number	email	address	bill_value	card_number	card_holder	date_value	cvc
1	John	999999999	JohnDoe@gmail.com	New jersey, USA	99	123456789090	John Doe	12/24	111
4	Umasree Kollu	123456789	umasree@example.com	123 Main Street	100	1234 5678 9012 3456	Umasree Kollu	12/24	123
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

#	Time	Action	Message	Duration / Fetch
2	21:46:05	SELECT * FROM products.payment	4 row(s) returned	0.000 sec / 0.000 sec
3	21:46:29	SELECT * FROM products.payment	2 row(s) returned	0.000 sec / 0.000 sec

Payment details in Database

Conclusion:

Integrating payment functionality in Spring Boot allows seamless transactions. Testing with Postman ensures smooth operation, while MySQL storage ensures data persistence. Efficiently implementing these steps enhances application reliability and user experience.

Thank you for reading this blog post! I hope you found the information helpful and insightful. If you have any questions or feedback, please feel free to reach out. Happy coding!

Payment Gateway Spring Framework Spring Boot

Follow

Written by Umasree Kollu

17 Followers

More from Umasree Kollu

Umasree Kollu

Integrating Stripe payments in Spring Boot

Discover how to seamlessly integrate Stripe, a leading payment gateway, into your Spring Boot application. Simplify payment processing and...

Apr 8

See all from Umasree Kollu

Recommended from Medium

Sudhakarketha

Integrating Stripe payment with Spring Boot

Discover how to seamlessly integrate Stripe, a leading payment gateway, into your Spring Boot application. Simplify payment processing and...

Apr 9 6 1

SumitM

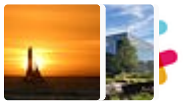
10 Spring Boot Interview Questions for Beginners

Spring Boot is a powerful framework used for developing production-ready, Java-based applications with ease. It simplifies configuration...

5d ago 20 2

Lists

Staff Picks
742 stories · 1332 saves


Stories to Help You Level-Up at Work
19 stories · 815 saves

Self-Improvement 101
20 stories · 2807 saves

Productivity 101
20 stories · 2394 saves

Master Spring Ter

Implementing Rate Limiting in Spring Boot with Bucket4j

As web applications grow and handle more traffic, ensuring their reliability and performance becomes crucial. One effective way to achieve...

Jun 16 7

Vinotech

Pessimistic Locking in JPA with Spring Boot

Understanding Pessimistic Locking

Sep 1 2 1

Rabinarayan Patra in Insights from ThoughtClan

The Best Way to Structure API Responses in Spring Boot

Discover the best practices for structuring API responses in Spring Boot. Enhance clarity, consistency, and maintainability in your APIs...

Aug 2 341 13

DesignNerds in Dev Genius

Low level design: Payment gateway system

Designing a Robust Payment Gateway System: Class Structure, Error Handling, and Transaction Rollback

6d ago 19

See more recommendations