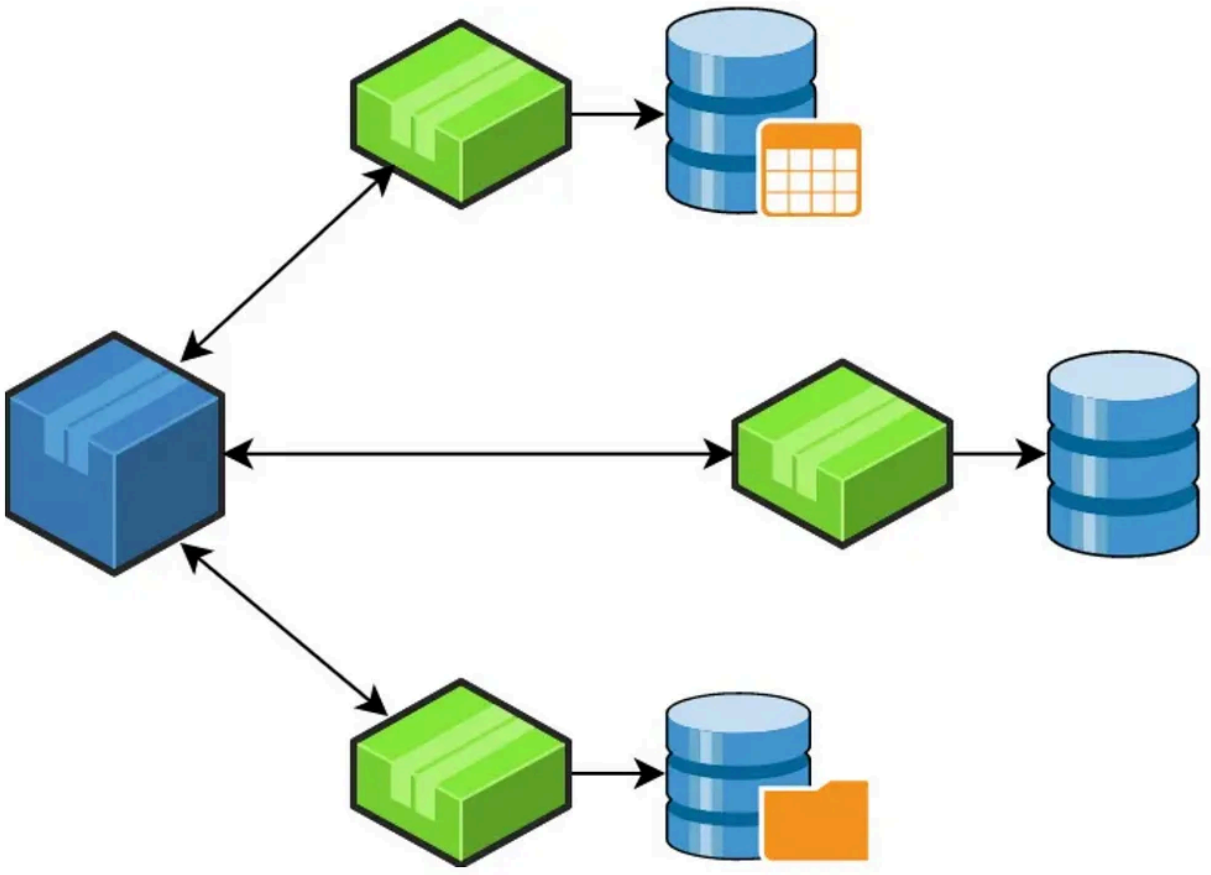
Distributed Transactions in Spring Boot Microservices: Simplified Guide

a anudeep billa · Follow
10 min read · Dec 20, 2023



Img Src: https://miro.medium.com/v2/resize:fit:690/1*ZbA4HrE9XKF4FziPs2MNfQ.png

Have you ever wondered how complex applications like e-commerce websites or banking systems manage to execute numerous operations simultaneously without error, especially when these operations span across different services and databases? The key to this orchestration lies in understanding what a transaction is and how distributed transactions are managed in a microservices architecture.

What is a Transaction?

At its core, a transaction in software systems is a group of operations that are executed as a single unit. This concept is crucial for maintaining data integrity and consistency, particularly in database operations. The most classic example of a transaction can be found in the banking sector. Consider a simple fund transfer from one account to another. This operation consists of two main steps: debiting an amount from one account and crediting it to another. These two operations together form a single transaction. The critical aspect here is that both operations must either be completed successfully or not executed at all. If the debit operation is successful but the credit operation fails due to some error, the transaction must be rolled back, reverting the debit to maintain the balance in the accounts. This is where the ACID properties of transactions come into play:

- **Atomicity**: This property ensures that all operations within a transaction are treated as a single unit. The transaction either fully happens or doesn't happen at all.
- **Consistency**: This guarantees that a transaction can only bring the system from one valid state to another, maintaining database rules and constraints.

- **Isolation**: Transactions are often executed concurrently. Isolation ensures that concurrent transactions do not interfere with each other.
- **Durability**: Once a transaction is committed, it remains so, even in case of system failures. This ensures that the results of the transaction are permanently recorded

Transactions in Monolithic Architecture vs. Microservices Architecture

Understanding the differences in transaction management between monolithic and microservices architectures is crucial for developers and architects. These differences highlight how various architectural designs influence the approach to ensuring data consistency and integrity in software applications.

Transactions in Monolithic Architecture

In a monolithic architecture, the application is developed as a single, indivisible unit. This includes its database interactions, business logic, and user interface components. Transactions in such a setup are typically straightforward to manage due to the centralized nature of the data and services.

Example:

Consider a simple online bookstore in a monolithic architecture. When a user makes a purchase, several steps occur:
1. The inventory is checked and updated.
2. The payment is processed.
3. The order is recorded.

These steps are executed as part of a single transaction in the same database. If any step fails, the entire transaction is rolled back to maintain data consistency. This rollback is easy to manage since all the steps are tightly integrated and interact with a single database system.

The ACID (Atomicity, Consistency, Isolation, Durability) properties are inherently maintained in this architecture. The monolithic structure ensures that transactions are atomic, the database remains consistent after transactions, operations are isolated, and changes are durable.

Transactions in Microservices Architecture

Microservices architecture breaks down an application into smaller, independent services, each with its database. This distributed nature poses significant challenges in maintaining transactional integrity across different services and databases.

Example:

Let's transform the online bookstore into a microservices architecture. The application is now split into separate services: Inventory Service, Payment Service, and Order Service. Each service has its database.

When a user places an order, the transaction spans across these services:
1. The Inventory Service checks and updates the stock.
2. The Payment Service processes the payment.
3. The Order Service records the order.

In this scenario, managing the transaction is more complex. If the payment processing fails after the inventory is updated, we need a mechanism to revert the inventory update to maintain data consistency. Unlike the

monolithic architecture, we can't rely on a single database transaction rollback.

How to manage the transaction isolation level for concurrent requests?

We can address these challenges with the help of @Transcational Annotation, a two-phase commit (2PC) pattern or SAGA pattern. Let's try to understand each in a bit more detail

Declarative Transaction Management

This approach is one of the main highlights of Spring's transaction capabilities. It promotes a cleaner, POJO-driven methodology, allowing developers to annotate methods that need to be executed within transaction boundaries.

- **@Transactional Annotation:** The heart of declarative transaction management. By annotating a method with `@Transactional`, Spring ensures that the method is executed within a transactional context. This simplifies the developer's role, focusing more on business logic rather than infrastructural concerns.

```
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.beans.factory.annotation.Autowired;

@Service
public class BookstoreService {

    @Autowired
    private InventoryService inventoryService;
    @Autowired
    private PaymentService paymentService;
```

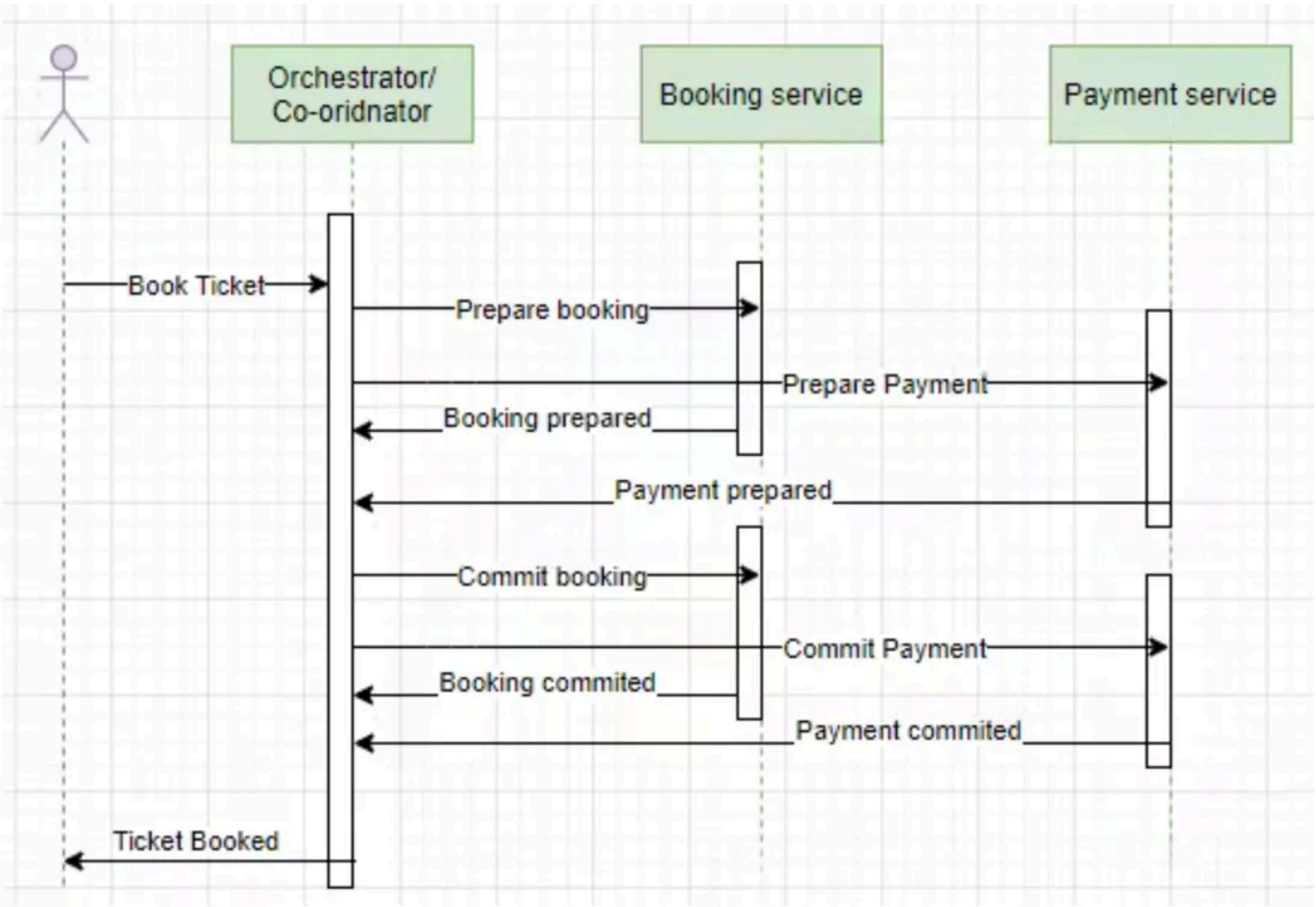
```
    @Autowired
    private OrderRepository orderRepository;

    @Transactional
    public void processOrder(Order order) {
        // Step 1: Check and update inventory
        boolean inventoryUpdated = inventoryService.updateInventory(order.getBoo
        if (!inventoryUpdated) {
            // If inventory update fails, transaction will be automatically roll
            throw new RuntimeException("Failed to update inventory for book: " +
        }

        // Step 2: Process payment
        boolean paymentProcessed = paymentService.processPayment(order.getPaymen
        if (!paymentProcessed) {
            // If payment processing fails, transaction will be automatically ro
            throw new RuntimeException("Payment processing failed for order: " +
        }

        // Step 3: Record the order
        orderRepository.save(order);
    }
}
```

- `@Transactional` Annotation: By annotating the `processOrder` method with `@Transactional`, we tell Spring to manage this method as a transaction. This means that all the operations within this method are either completed successfully or none at all.
- **Automatic Rollback:** If any operation within this method fails (e.g., inventory update fails or payment processing fails), Spring's transaction management will automatically roll back any changes made up to the point of failure. This ensures data consistency and integrity.
- **Simplified Workflow:** The primary advantage here is the simplicity of managing complex workflows. The developer focuses on the business logic, while Spring handles the transaction boundaries and rollback mechanisms.

Two-Phase Commit — 2PC

The Two-Phase Commit is a distributed algorithm that allows multiple services to agree on a common outcome in a transaction. It is mainly used to ensure atomicity across a distributed system. The pattern involves two phases:

1. **Prepare Phase:** Each participant (service) prepares to complete the transaction and votes either to commit or abort.
2. **Commit Phase:** Based on the voting in the prepare phase, if all services are ready to commit, the coordinator initiates the commit phase. If any service votes to abort, the transaction is rolled back across all services.



In this example, when a user attempts to book a ticket, the Orchestrator begins the 2PC process to ensure both the booking and payment are either completed or both are aborted, maintaining the atomicity of the transaction across the distributed system.

The process is as follows:

1. The Orchestrator sends a request to the Booking Service to prepare for booking. The Booking Service ensures it can reserve the ticket and responds affirmatively.
2. Concurrently, the Orchestrator sends a request to the Payment Service to prepare for payment. The Payment Service verifies payment can be processed and responds that it's prepared.
3. Upon receiving affirmative responses from both services, the Orchestrator sends a message to commit the booking. The Booking Service finalizes the reservation and confirms the booking commitment.
4. The Orchestrator then sends a message to the Payment Service to commit the payment. The Payment Service processes the payment and confirms the transaction commitment.
5. Finally, the Orchestrator concludes the transaction process, ensuring the user that the ticket has been booked.

Challenges with 2PC

Implementing 2PC in Spring microservices can pose several challenges:

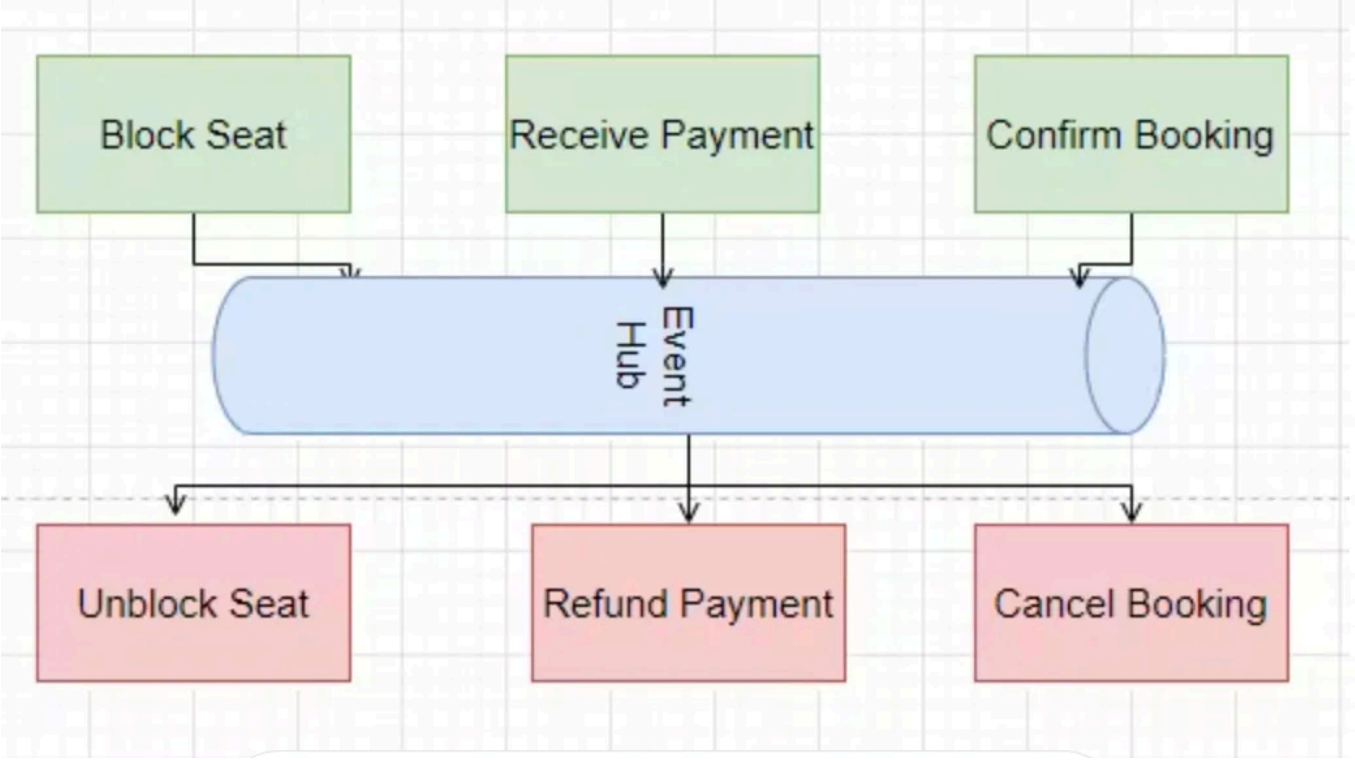
- **Synchronous Blocks**: The 2PC pattern requires services to wait during the preparation phase, which can lead to synchronous blocks and increased response times.

- **Single Point of Failure**: The orchestrator/coordinator becomes a single point of failure. If it crashes or becomes unavailable, the entire transaction can be left in an indeterminate state.
- **Data Inconsistency:** In the event of a service failure during the commit phase, it can be challenging to ensure that all services roll back to their previous state, leading to potential data inconsistency.
- **Complex Error Handling:** Handling errors and implementing rollback mechanisms in microservices can be complex due to the distributed nature of the services.
- **Resource Locking:** During the prepare phase, resources may be locked, which can reduce system performance and responsiveness

In a microservice architecture, especially one using Spring, the 2PC pattern is less favored due to these challenges. Instead, patterns like Saga are often preferred, as they offer a way to handle distributed transactions without the need for tight coordination and locking, thus providing better resilience and performance in distributed systems.

Saga Pattern

The Saga pattern is a sequence of local transactions, where each service performs its part of the process. If any step fails, compensating transactions are executed to undo the previous steps, ensuring data consistency without the need for a two-phase commit.



In the given scenario, the process to book a seat for an event involves three key steps: blocking the seat, receiving payment, and confirming the booking. These steps are likely handled by different microservices. Here's how the Saga pattern applies:

1. **Block Seat:** The seat reservation service attempts to block a seat. If successful, it sends an event message to the next service.
2. **Receive Payment:** The payment service listens for the seat blocked event, attempts to process the payment, and upon success, sends an event message to confirm the booking.
3. **Confirm Booking:** The booking service listens for the payment received event and finalizes the booking, confirming the seat reservation.

If any step in this sequence fails, for example, if the payment processing fails after the seat has been blocked, compensating transactions are triggered:

- **Refund Payment:** If payment was initially recorded but failed afterward, the payment service would issue a refund.
- **Unblock Seat:** In response to payment failure, the seat reservation service would unblock the seat.
- **Cancel Booking:** Similarly, if the booking had been initially noted but could not be confirmed, the booking service would cancel the booking.

This series of compensating transactions ensures that the system remains consistent and that all services revert to their initial state before the Saga began.

Sample Pusedocode

```
@Service
public class BookingSagaService {

    @Autowired
    private SeatReservationService seatReservationService;

    @Autowired
    private PaymentService paymentService;

    @Autowired
    private BookingService bookingService;

    public void executeSaga(Order order) {
        try {
            // Step 1: Block Seat
            seatReservationService.blockSeat(order.getEventId(), order.getSeatId
            // Step 2: Receive Payment
            paymentService.processPayment(order.getPaymentDetails());
            // Step 3: Confirm Booking
            bookingService.confirmBooking(order);
        } catch (SeatReservationException | PaymentProcessingException | Booking
            // Handle compensating transactions
            compensateTransaction(order);
            throw e;
```

```
            }
        }

        private void compensateTransaction(Order order) {
            // Compensate Block Seat
            seatReservationService.unblockSeat(order.getEventId(), order.getSeatId()
            // Compensate Receive Payment
            paymentService.refundPayment(order.getPaymentDetails());
            // Compensate Confirm Booking
            bookingService.cancelBooking(order);
        }
    }

    // Exceptions for illustration
    class SeatReservationException extends RuntimeException { /* ... */ }
    class PaymentProcessingException extends RuntimeException { /* ... */ }
    class BookingException extends RuntimeException { /* ... */ }

    // Placeholder methods for services
    class SeatReservationService {
        void blockSeat(String eventId, String seatId) { /* ... */ }
        void unblockSeat(String eventId, String seatId) { /* ... */ }
    }

    class PaymentService {
        void processPayment(PaymentDetails paymentDetails) { /* ... */ }
        void refundPayment(PaymentDetails paymentDetails) { /* ... */ }
    }

    class BookingService {
        void confirmBooking(Order order) { /* ... */ }
        void cancelBooking(Order order) { /* ... */ }
    }

    // Placeholder domain classes
    class Order {
        private String eventId;
        private String seatId;
        private PaymentDetails paymentDetails;
        // Getters and setters
    }

    class PaymentDetails {
        // Payment details fields
        // Getters and setters
    }
```

- Each service (`SeatReservationService` , `PaymentService` , and `BookingService`) has a method to perform its transaction as well as a compensating method to undo it.
- If an exception is thrown at any point during the saga execution, the `compensateTransaction` method is called to execute the compensating transactions to rollback the changes made up to that point.
- The `executeSaga` method orchestrates the entire booking process, ensuring that if any part fails, the system remains in a consistent state.

Conclusion

Spring Boot offers a suite of tools tailored for handling distributed transactions within a microservice ecosystem, accommodating both the Two-Phase Commit (2PC) and Saga transaction patterns. These patterns provide mechanisms to maintain consistency across disparate services within a distributed system.

The 2PC pattern is a conventional technique that employs a centralized coordinator to govern transactions across various services. This method guarantees atomicity, where all participating services either commit or abort changes together, thus ensuring data consistency across the system. Despite its robustness in maintaining consistency, the 2PC approach presents several drawbacks when applied to microservices. It can introduce performance bottlenecks due to the synchronous nature of its operations, increase system complexity, and create potential single points of failure. Moreover, resource locking during the transaction can impede the scalability and resilience that microservices aim to provide.

In contrast, the Saga pattern emerges as a more decentralized approach, where transactions are broken down into a series of local transactions. Each

service handles its segment of the overall process, and in the case of failures, compensating transactions are triggered to revert the system to its initial state. This pattern aligns more naturally with the distributed and loosely coupled nature of microservices, offering improved fault tolerance and system responsiveness.

Spring Boot also simplifies transaction management within individual microservices through the declarative '`@Transactional`'annotation. This allows developers to define transaction boundaries directly in the application code, leaving the runtime to manage transactional complexities such as rollbacks and commits. The '`@Transactional`'annotation is particularly beneficial in maintaining cleaner and more maintainable service methods, as it abstracts away the boilerplate code associated with programmatic transaction management. However, it's important to note that '`@Transactional`'is typically used for local transactions within a service and does not directly manage distributed transactions across services.

In summary, Spring Boot's support for 2PC and Saga patterns, combined with the convenience of the '`@Transactional`'annotation, equips developers with a robust set of options for managing transactions. By understanding the characteristics and use cases for each pattern, as well as the advantages of Spring's transactional annotations, developers can make informed decisions to architect resilient and consistent microservice applications.

References :

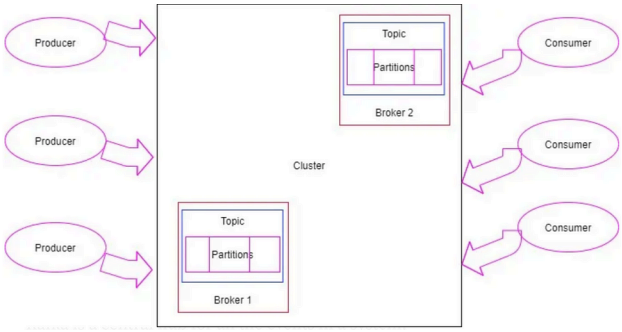
https://blog.knowledge-cafe.dev/distributed-transaction-in-spring-boot-microservices

Written by anudeep billa

20 Followers

Follow

More from anudeep billa



anudeep billa

Intro to Apache Kafka with Microservices in SpringBoot

The Genesis of Kafka

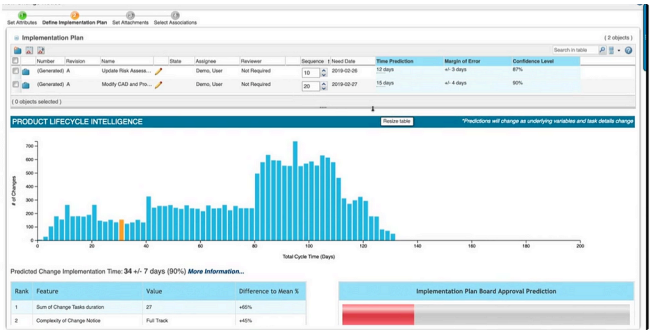
Oct 27, 2023 6

anudeep billa

Concept to Code: Our Silver Finish at HackNC2023

Setting the Stage: The Electric Atompshere of HackNC

Nov 1, 2023



anudeep billa

PLM In Data Analytics Perspective.

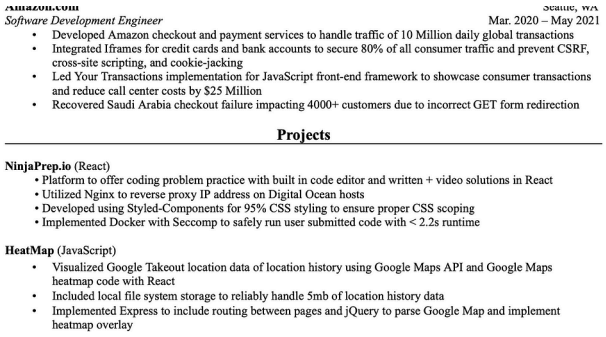
In the era of Industry 4.0 , product lifecycle management is evolving into a new paradig...

Oct 28, 2021 1

See all from anudeep billa

Recommended from Medium

Oliver Foster



Software Development Engineer Mar. 2020 – May 2021

- Developed Amazon checkout and payment services to handle traffic of 10 Million daily global transactions
- Integrated Iframes for credit cards and bank accounts to secure 80% of all consumer traffic and prevent CSRF, cross-site scripting, and cookie-jacking
- Led Your Transactions implementation for JavaScript front-end framework to showcase consumer transactions and reduce call center costs by $25 Million
- Recovered Saudi Arabia checkout failure impacting 4000+ customers due to incorrect GET form redirection

Projects

NinjaPrep.io (React)
- Platform to offer coding problem practice with built in code editor and written + video solutions in React
- Utilized Nginx to reverse proxy IP address on Digital Ocean hosts
- Developed using Styled-Components for 95% CSS styling to ensure proper CSS scoping
- Implemented Docker with Seccomp to safely run user submitted code with < 2.2s runtime

HeatMap (JavaScript)
- Visualized Google Takeout location data of location history using Google Maps API and Google Maps heatmap code with React
- Included local file system storage to reliably handle 5mb of location history data
- Implemented Express to include routing between pages and jQuery to parse Google Map and implement heatmap overlay

Alexander Nguyen in Level Up Coding

Spring Boot: How Many Requests Can Spring Boot Handle...

My article is open to everyone; non-member readers can click this link to read the full text.

Jun 17 824 12

The resume that got a software engineer a $300,000 job at Google.

1-page. Well-formatted.

Jun 1 11.8K 156

Lists

Staff Picks

684 stories · 1124 saves

Stories to Help You Level-Up at Work

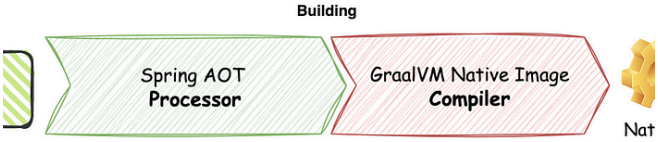
19 stories · 685 saves

Self-Improvement 101

20 stories · 2270 saves

Productivity 101

20 stories · 2001 saves



Saeed Zarinfam in ITNEXT

10 Spring Boot Performance Best Practices

Making Spring Boot applications performant and resource-efficient

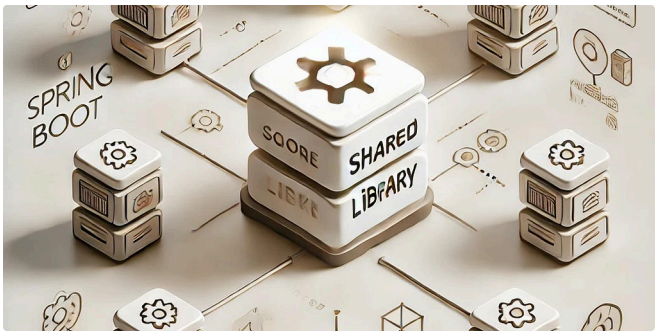
Jun 24 183 3

Renan Schmitt in JavaJams

Spring Boot: Handling a REST Endpoint That Queries More Data...

If you are a developer and have not faced this issue yet, it is still worth reading, as at some...

Jun 24 230 4

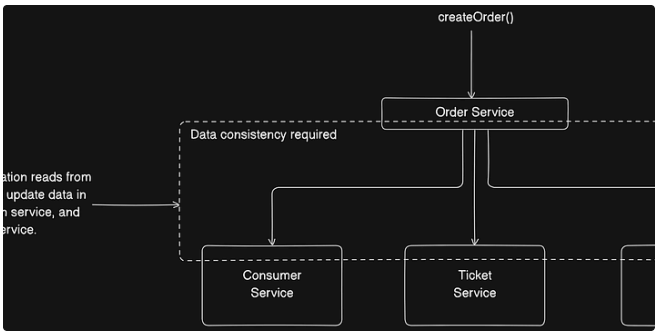


Vasko Jovanoski

Say Goodbye to Repetition: Building a Common Library in...

Learn to create a shared library in Spring Boot, enhancing code reusability, simplifying...

Jun 25 257 8



Joud W. Awad

Microservices Pattern: Distributed Transactions (SAGA)

Explore the SAGA Pattern: Ensuring Data Integrity in Microservices. Dive into its...

Jun 17 560 4

See more recommendations