

Kestra in Lostigisic Systems - 1320d9185b

USER

in a lostigisic system where we can use kestra

BOT

> Routing to GPT-4o

Kestra is an open-source orchestration and scheduling platform designed to manage complex workflows and data pipelines. In a logistics system, Kestra can be particularly useful for automating and optimizing various processes. Here are some ways Kestra can be applied in a logistics context:

1. **Order Processing Automation**:
 - Automate the entire order processing workflow, from order receipt to fulfillment.
 - Integrate with inventory management systems to update stock levels in real-time.
2. **Supply Chain Management**:
 - Coordinate between different stages of the supply chain, ensuring timely communication and data exchange.
 - Automate notifications and alerts for supply chain disruptions or delays.
3. **Route Optimization**:
 - Schedule and optimize delivery routes based on real-time traffic data and delivery priorities.
 - Integrate with GPS and mapping services to provide dynamic routing solutions.
4. **Inventory Management**:
 - Automate inventory checks and reorder processes based on predefined thresholds.
 - Integrate with suppliers to automate purchase orders and track shipments.
5. **Data Integration and ETL**:
 - Extract, transform, and load (ETL) data from various sources to provide a unified view of logistics operations.
 - Automate data synchronization between different systems, such as ERP, CRM, and WMS.
6. **Monitoring and Reporting**:
 - Set up automated monitoring of logistics KPIs and generate reports for performance analysis.
 - Trigger alerts for any anomalies or deviations from expected performance metrics.
7. **Customer Notifications**:
 - Automate customer notifications for order status updates, delivery schedules, and any changes.
 - Integrate with communication platforms to send notifications via email, SMS, or other channels.

By leveraging Kestra, logistics companies can enhance their operational efficiency, reduce manual intervention, and improve overall service quality. The platform's ability to integrate with various systems and automate complex workflows makes it a valuable tool in the logistics industry.

USER

i need start with it so from where we can start

BOT

> Routing to GPT-4o Mini

Getting started with Kestra involves several steps, from understanding its architecture to deploying your first workflow.

Here's a structured approach to help you begin:

1. **Understand the Basics of Kestra**

- **Documentation**: Start by reading the [official Kestra documentation](https://kestra.io/docs/). This will give you a solid understanding of its features, architecture, and components.
- **Key Concepts**: Familiarize yourself with key concepts such as **tasks**, **workflows**, **executions**, and **plugins**.

2. **Set Up Your Environment

- **System Requirements**: Ensure your system meets the requirements for running Kestra. It can be run locally or deployed on a server.
- **Installation**:
 - **Docker**: The easiest way to get started is by using Docker. You can pull the official Kestra Docker image and run it locally.
 - **Kubernetes**: For production environments, consider deploying Kestra on Kubernetes.

Example Docker Command:

```
```bash
docker run -p 8080:8080 -e KESTRA_SERVER_URL=http://localhost:8080 -e
KESTRA_DATABASE_URL=jdbc:postgresql://localhost:5432/kestra -e KESTRA_DATABASE_USER=kestra -e
KESTRA_DATABASE_PASSWORD=kestra kestra/kestra:latest
```
```

3. **Create Your First Workflow

- **Define a Workflow**: Start by creating a simple workflow. You can define workflows in YAML format. Here's a basic example:

```
```yaml
id: my-first-workflow
namespace: default
tasks:
 - id: my-task
 type: io.kestra.core.tasks.scripts.Bash
 commands:
 - echo "Hello, Kestra!"
```
```

- **Upload the Workflow**: Use the Kestra UI or API to upload your workflow definition.

4. **Explore the User Interface

- Access the Kestra UI (usually at `http://localhost:8080` if running locally) to manage your workflows, view executions, and monitor logs.
- Familiarize yourself with the dashboard, where you can see the status of your workflows and tasks.

5. **Learn About Plugins and Integrations

- Kestra supports various plugins for different tasks (e.g., HTTP requests, database operations, etc.). Explore the available plugins in the documentation and experiment with them in your workflows.

6. **Experiment and Iterate

- Start building more complex workflows by chaining tasks together, using conditions, and integrating with external systems.
- Test your workflows thoroughly to ensure they behave as expected.

7. **Join the Community

- Engage with the Kestra community through forums, GitHub, or Discord. This can be a great way to get support, share ideas, and learn from others.

8. **Advanced Topics

- Once you're comfortable with the basics, explore advanced topics such as:
 - **Error Handling**: Learn how to manage errors and retries in your workflows.

- **Scheduling**: Set up schedules for your workflows to run at specific times.
- **Monitoring and Alerts**: Implement monitoring and alerting for your workflows.

Resources

- **Official Documentation**: [Kestra Documentation](https://kestra.io/docs/)
- **GitHub Repository**: [Kestra GitHub](https://github.com/kestra-io/kestra)
- **Community Support**: Check out forums or Discord channels related to Kestra for community support.

By following these steps, you should be well on your way to effectively using Kestra in your logistics system or any other application you have in mind. If you have specific questions or need further assistance, feel free to ask!

USER

there is two system in our company

- 1). booking for the logistics
- 2). Delivery for Logistics

so i need to create a whole new system for the booking for the logistics

so give me the whole achitecture how we can use kestra

BOT

> Routing to GPT-4o

Designing a new system for booking logistics using Kestra involves creating an architecture that integrates various components to handle tasks such as order processing, scheduling, and data management. Below is a high-level architecture that outlines how you can use Kestra to build a booking system for logistics:

High-Level Architecture

1. **User Interface (UI)**
 - **Web Application**: Develop a web application for users to book logistics services. This can be built using frameworks like React, Angular, or Vue.js.
 - **Mobile Application**: Optionally, create a mobile app for on-the-go booking.
2. **Backend Services**
 - **API Gateway**: Use an API gateway to manage and route requests from the UI to backend services.
 - **Booking Service**: A microservice responsible for handling booking requests, validating data, and interacting with the database.
 - **Notification Service**: A service to send notifications (email, SMS) to customers about booking confirmations and updates.
3. **Kestra Orchestration**
 - **Workflow Management**: Use Kestra to define and manage workflows for booking processes. This includes tasks like:
 - Validating booking details.
 - Checking availability.
 - Confirming bookings.
 - Integrating with the delivery system for scheduling.
 - **Task Automation**: Automate repetitive tasks such as sending confirmation emails, updating inventory, and generating invoices.
4. **Database**
 - **Relational Database**: Use a relational database (e.g., PostgreSQL, MySQL) to store booking information, customer data, and transaction records.
 - **NoSQL Database**: Optionally, use a NoSQL database (e.g., MongoDB) for handling unstructured data or high-volume data storage.
5. **Integration Layer**
 - **API Integration**: Integrate with external systems (e.g., payment gateways, third-party logistics providers) through APIs.
 - **Data Sync**: Use Kestra to automate data synchronization between the booking system and the delivery system.

6. **Monitoring and Logging**

- **Monitoring Tools**: Implement monitoring tools (e.g., Prometheus, Grafana) to track system performance and health.
- **Logging**: Use centralized logging (e.g., ELK Stack) to capture and analyze logs for troubleshooting and auditing.

7. **Security**

- **Authentication and Authorization**: Implement secure authentication (e.g., OAuth2, JWT) and role-based access control.
- **Data Encryption**: Ensure data is encrypted both in transit and at rest.

8. **Deployment and Scalability**

- **Containerization**: Use Docker to containerize services for consistent deployment.
- **Orchestration**: Deploy services using Kubernetes for scalability and resilience.
- **CI/CD Pipeline**: Set up a CI/CD pipeline for automated testing and deployment.

Workflow Example in Kestra

Here's a simplified example of a Kestra workflow for processing a booking:

```
```yaml
id: booking-process
namespace: logistics
tasks:
 - id: validate-booking
 type: io.kestra.core.tasks.scripts.Bash
 commands:
 - echo "Validating booking details..."

 - id: check-availability
 type: io.kestra.core.tasks.scripts.Bash
 commands:
 - echo "Checking availability..."

 - id: confirm-booking
 type: io.kestra.core.tasks.scripts.Bash
 commands:
 - echo "Confirming booking..."

 - id: send-confirmation
 type: io.kestra.core.tasks.scripts.Bash
 commands:
 - echo "Sending confirmation email..."
```
```

Steps to Implement

1. **Define Requirements**: Gather detailed requirements for the booking system, including user stories and business rules.
2. **Design Architecture**: Create detailed architecture diagrams and design documents.
3. **Develop Components**: Implement the UI, backend services, and Kestra workflows.
4. **Integrate Systems**: Ensure seamless integration with the delivery system and other external services.
5. **Test and Deploy**: Conduct thorough testing and deploy the system using a CI/CD pipeline.
6. **Monitor and Optimize**: Continuously monitor the system and optimize workflows and services for performance and reliability.

By following this architecture, you can build a robust and scalable booking system for logistics that leverages Kestra for workflow automation and orchestration. If you need further details on any specific component or process, feel free to ask!

USER

BOT

> Routing to Claude Sonnet 3.5

Logistics Booking System Implementation Plan

Phase 1: Project Setup and Initial Planning (4-6 weeks)

1.1 Requirements Gathering

- [] Document detailed functional requirements
- [] Create user stories and use cases
- [] Define system boundaries and constraints
- [] Identify integration points with existing delivery system
- [] Document non-functional requirements (performance, scalability, security)

1.2 Architecture Design

- [] Create detailed system architecture diagrams
- [] Design database schema
- [] Define API specifications
- [] Plan security architecture
- [] Design monitoring and logging strategy

1.3 Technology Stack Selection

- Frontend: React.js with TypeScript
- Backend: Node.js/Spring Boot
- Database: PostgreSQL
- Message Queue: RabbitMQ
- Orchestration: Kestra
- Container: Docker
- Container Orchestration: Kubernetes
- CI/CD: Jenkins/GitLab CI

Phase 2: Development Environment Setup (2-3 weeks)

2.1 Infrastructure Setup

- [] Set up development environment
- [] Configure version control (Git)
- [] Set up CI/CD pipeline
- [] Configure Docker registry
- [] Set up Kubernetes cluster
- [] Install and configure Kestra

2.2 Database Setup

- [] Set up PostgreSQL database
- [] Create initial schema
- [] Set up database backup and recovery procedures
- [] Configure database monitoring

Phase 3: Core Development (12-16 weeks)

3.1 Backend Development

- [] Implement API Gateway

- [] Develop core booking service
- [] Create authentication and authorization system
- [] Implement notification service
- [] Develop integration services

3.2 Frontend Development

- [] Create responsive UI design
- [] Implement booking interface
- [] Develop dashboard for tracking bookings
- [] Create admin panel
- [] Implement user management interface

3.3 Kestra Workflow Implementation

This comprehensive plan provides a structured approach to implementing the logistics booking system. Would you like me to elaborate on any specific phase or component? We can also create more detailed technical specifications for any particular aspect of the system.

Some key areas we could dive deeper into:

1. Detailed database schema design
2. Specific Kestra workflow implementations
3. API specifications
4. Infrastructure architecture
5. Security implementation details

Let me know which area you'd like to explore further!

USER

make HLD and LLD diagram in mairmaid

BOT

> Routing to Claude Sonnet 3.5

```
graph TB
  subgraph Client_Layer [Client Layer]
    Web[Web Application]
    Mobile[Mobile Application]
    Admin[Admin Portal]
  end

  subgraph API_Gateway [API Gateway]
    Gateway[API Gateway/Load Balancer]
  end

  subgraph Microservices [Microservices]
    BS[Booking Service]
    AS[Authentication Service]
    NS[Notification Service]
    PS[Payment Service]
  end
```

```
subgraph Kestra Orchestration
```

```
  KO[Kestra Orchestrator]
```

```
  WF[Workflow Engine]
```

```
  TM[Task Manager]
```

```
end
```

```
subgraph Data Layer
```

```
  DB[(PostgreSQL)]
```

```
  Cache[(Redis Cache)]
```

```
  MQ[RabbitMQ]
```

```
end
```

```
subgraph External Systems
```

```
  DS[Delivery System]
```

```
  PS[Payment Gateway]
```

```
  ES[Email Service]
```

```
end
```

```
Web --> Gateway
```

```
Mobile --> Gateway
```

```
Admin --> Gateway
```

```
Gateway --> BS
```

```
Gateway --> AS
```

```
Gateway --> NS
```

```
BS --> KO
```

```
KO --> WF
```

```
WF --> TM
```

```
BS --> DB
```

```
BS --> Cache
```

```
BS --> MQ
```

```
KO --> DS
```

```
KO --> PS
```

```
NS --> ES
```

```
...
```

```
...
```

```
graph TB
```

```
  subgraph Client Layer
```

```
    Web[Web Application]
```

```
    Mobile[Mobile Application]
```

```
    Admin[Admin Portal]
```

```
  end
```

```
  subgraph API Gateway
```

```
    Gateway[API Gateway/Load Balancer]
```

```
  end
```

```
  subgraph Microservices
```

```
    BS[Booking Service]
```

```
    AS[Authentication Service]
```

```
    NS[Notification Service]
```

```
    PS[Payment Service]
```

```
  end
```

```
  subgraph Kestra Orchestration
```

```
    KO[Kestra Orchestrator]
```

```
    WF[Workflow Engine]
```

```
    TM[Task Manager]
```

end

```
subgraph Data Layer
  DB[(PostgreSQL)]
  Cache[(Redis Cache)]
  MQ[RabbitMQ]
end
```

```
subgraph External Systems
  DS[Delivery System]
  PS[Payment Gateway]
  ES[Email Service]
end
```

```
Web --> Gateway
Mobile --> Gateway
Admin --> Gateway
Gateway --> BS
Gateway --> AS
Gateway --> NS
```

```
BS --> KO
KO --> WF
WF --> TM
```

```
BS --> DB
BS --> Cache
BS --> MQ
```

```
KO --> DS
KO --> PS
NS --> ES
```

Let me explain the key components of both diagrams:

High Level Design (HLD) Explanation:

1. ****Client Layer****: Contains all user interfaces (Web, Mobile, Admin Portal)
2. ****API Gateway****: Central entry point for all client requests
3. ****Microservices****: Core services handling different business functionalities
4. ****Kestra Orchestration****: Workflow management and task orchestration
5. ****Data Layer****: Data storage and message queuing
6. ****External Systems****: Third-party integrations

Low Level Design (LLD) Explanation:

1. ****BookingController****: REST API endpoints for booking operations
2. ****BookingService****: Core business logic implementation
3. ****BookingRepository****: Data access layer
4. ****Domain Models****:
 - Booking: Main entity for booking information
 - Customer: Customer details
 - BookingDetails: Specific booking requirements
5. ****KestraWorkflow****: Workflow management implementation

The diagrams show:

- Clear separation of concerns
- Microservices architecture
- Data flow between components
- Entity relationships
- Service boundaries

Would you like me to:

1. Add more detail to any specific component?
2. Create additional diagrams for other subsystems?
3. Explain any particular flow in more detail?

USER

diagrams looks good but need animated and more clear

BOT

> Routing to Claude Sonnet 3.5

```
graph TB
%% Styling
classDef clientLayer fill:#e1f5fe,stroke:#01579b,stroke-width:2px;
classDef gateway fill:#fff3e0,stroke:#ff6f00,stroke-width:2px;
classDef microservice fill:#e8f5e9,stroke:#2e7d32,stroke-width:2px;
classDef kestra fill:#f3e5f5,stroke:#7b1fa2,stroke-width:2px;
classDef dataLayer fill:#fbee9e,stroke:#d84315,stroke-width:2px;
classDef external fill:#efebee,stroke:#3e2723,stroke-width:2px;

%% Client Layer
subgraph CL[Client Layer]
    Web[Web Application]
    Mobile[Mobile Application]
    Admin[Admin Portal]
end

%% API Gateway
subgraph AG[API Gateway Layer]
    Gateway[API Gateway/Load Balancer]
    Auth[Authentication & Authorization]
end

%% Microservices
subgraph MS[Microservices Layer]
    BS[Booking Service]
    NS[Notification Service]
    PS[Payment Service]
    TS[Tracking Service]
end

%% Kestra Orchestration
subgraph KO[Kestra Orchestration]
    Orchestrator[Kestra Orchestrator]
    WF[Workflow Engine]
    TM[Task Manager]
    Scheduler[Job Scheduler]
end

%% Data Layer
subgraph DL[Data Layer]
    DB[(PostgreSQL)]
    Cache[(Redis Cache)]
    MQ[(RabbitMQ)]
end

%% External Systems
subgraph ES[External Systems]
    DS[Delivery System]
    PG[Payment Gateway]
    Email[Email Service]
    SMS[SMS Gateway]
```

end

%% Connections with animations

Web & Mobile & Admin --> |REST API| Gateway

Gateway --> |Authenticate| Auth

Auth --> |Validated Requests| MS

BS --> |Orchestrate| Orchestrator

Orchestrator --> |Execute| WF

WF --> |Manage| TM

TM --> |Schedule| Scheduler

BS --> |Store| DB

BS --> |Cache| Cache

BS --> |Queue| MQ

NS --> |Send| Email & SMS

PS --> |Process| PG

BS --> |Track| DS

%% Apply styles

class Web,Mobile,Admin clientLayer;

class Gateway,Auth gateway;

class BS,NS,PS,TS microservice;

class Orchestrator,WF,TM,Scheduler kestra;

class DB,Cache,MQ dataLayer;

class DS,PG,Email,SMS external;

%% Click events

click Web "Handles web-based user interactions" _blank

click Mobile "Handles mobile app interactions" _blank

click BS "Manages booking operations" _blank

click Orchestrator "Coordinates workflows" _blank

click DB "Stores booking data" _blank

graph TB

%% Styling

classDef clientLayer fill:#e1f5fe,stroke:#01579b,stroke-width:2px;

classDef gateway fill:#fff3e0,stroke:#ff6f00,stroke-width:2px;

classDef microservice fill:#e8f5e9,stroke:#2e7d32,stroke-width:2px;

classDef kestra fill:#f3e5f5,stroke:#7b1fa2,stroke-width:2px;

classDef dataLayer fill:#fbee9e,stroke:#d84315,stroke-width:2px;

classDef external fill:#efeb3b,stroke:#3e2723,stroke-width:2px;

%% Client Layer

subgraph CL[Client Layer]

Web[Web Application]

Mobile[Mobile Application]

Admin[Admin Portal]

end

%% API Gateway

subgraph AG[API Gateway Layer]

Gateway[API Gateway/Load Balancer]

Auth[Authentication & Authorization]

end

%% Microservices

subgraph MS[Microservices Layer]

BS[Booking Service]

NS[Notification Service]

PS[Payment Service]

TS[Tracking Service]

end

```

%% Kestra Orchestration
subgraph KO[Kestra Orchestration]
  Orchestrator[Kestra Orchestrator]
  WF[Workflow Engine]
  TM[Task Manager]
  Scheduler[Job Scheduler]
end

%% Data Layer
subgraph DL[Data Layer]
  DB[(PostgreSQL)]
  Cache[(Redis Cache)]
  MQ[(RabbitMQ)]
end

%% External Systems
subgraph ES[External Systems]
  DS[Delivery System]
  PG[Payment Gateway]
  Email[Email Service]
  SMS[SMS Gateway]
end

%% Connections with animations
Web & Mobile & Admin --> |REST API| Gateway
Gateway --> |Authenticate| Auth
Auth --> |Validated Requests| MS
BS --> |Orchestrate| Orchestrator
Orchestrator --> |Execute| WF
WF --> |Manage| TM
TM --> |Schedule| Scheduler

BS --> |Store| DB
BS --> |Cache| Cache
BS --> |Queue| MQ

NS --> |Send| Email & SMS
PS --> |Process| PG
BS --> |Track| DS

%% Apply styles
class Web,Mobile,Admin clientLayer;
class Gateway,Auth gateway;
class BS,NS,PS,TS microservice;
class Orchestrator,WF,TM,Scheduler kestra;
class DB,Cache,MQ dataLayer;
class DS,PG,Email,SMS external;

%% Click events
click Web "Handles web-based user interactions" _blank
click Mobile "Handles mobile app interactions" _blank
click BS "Manages booking operations" _blank
click Orchestrator "Coordinates workflows" _blank
click DB "Stores booking data" _blank

```

These improved diagrams feature:

1. ****Clear Visual Hierarchy****:
 - Distinct color coding for different layers
 - Consistent styling for related components
 - Better spacing and organization

2. **Interactive Elements**:

- Clickable nodes with tooltips
- Hover effects for better visibility
- Clear relationship indicators

3. **Detailed Components**:

- Comprehensive service breakdown
- Clear data flow paths
- Explicit integration points

4. **Better Documentation**:

- Component descriptions
- Relationship explanations
- System boundaries

Would you like me to:

1. Add more detail to any specific component?
2. Create additional diagrams for specific workflows?
3. Add more interactive elements?
4. Include specific Kestra workflow diagrams?``

USER

how can we hit the kestra orkestratiion from the order booking service

BOT

> **Routing to Claude Sonnet 3.5**

```
@Service
public class BookingService {

    @Value("${kestra.api.url}")
    private String kestraApiUrl;

    @Autowired
    private RestTemplate restTemplate;

    public void processBooking(BookingRequest request) {
        // 1. Create booking in database
        Booking booking = createBooking(request);

        // 2. Prepare Kestra workflow input
        Map workflowInput = new HashMap<>();
        workflowInput.put("bookingId", booking.getId());
        workflowInput.put("customerId", booking.getCustomerId());
        workflowInput.put("deliveryAddress", booking.getDeliveryAddress());
        workflowInput.put("scheduledDate", booking.getScheduledDate());

        // 3. Trigger Kestra workflow
        triggerKestraWorkflow("logistics-booking-flow", workflowInput);
    }

    private void triggerKestraWorkflow(String workflowId, Map inputs) {
        String triggerUrl = kestraApiUrl + "/api/v1/executions/trigger";

        KestraExecutionRequest executionRequest = KestraExecutionRequest.builder()
            .namespace("logistics")
            .flowId(workflowId)
```

```

        .inputs(inputs)
        .build();

    try {
        ResponseEntity response = restTemplate.postForEntity(
            triggerUrl,
            executionRequest,
            KestraExecutionResponse.class
        );

        if (response.getStatusCode() == HttpStatus.OK) {
            log.info("Kestra workflow triggered successfully. Execution ID: {}",
                response.getBody().getExecutionId());
        }
    } catch (Exception e) {
        log.error("Failed to trigger Kestra workflow", e);
        throw new BookingProcessingException("Failed to initiate booking workflow", e);
    }
}

@Data
@Builder
class KestraExecutionRequest {
    private String namespace;
    private String flowId;
    private Map inputs;
}

@Data
class KestraExecutionResponse {
    private String executionId;
    private String state;
}

```

```

e;
}

```

```

...

```

```

...

```

```

@Service
public class BookingService {

    @Value("${kestra.api.url}")
    private String kestraApiUrl;

    @Autowired
    private RestTemplate restTemplate;

    public void processBooking(BookingRequest request) {
        // 1. Create booking in database
        Booking booking = createBooking(request);

        // 2. Prepare Kestra workflow input
        Map workflowInput = new HashMap<>();
        workflowInput.put("bookingId", booking.getId());
        workflowInput.put("customerId", booking.getCustomerId());
        workflowInput.put("deliveryAddress", booking.getDeliveryAddress());
        workflowInput.put("scheduledDate", booking.getScheduledDate());

        // 3. Trigger Kestra workflow
        triggerKestraWorkflow("logistics-booking-flow", workflowInput);
    }
}

```

```

private void triggerKestraWorkflow(String workflowId, Map inputs) {
    String triggerUrl = kestraApiUrl + "/api/v1/executions/trigger";

    KestraExecutionRequest executionRequest = KestraExecutionRequest.builder()
        .namespace("logistics")
        .flowId(workflowId)
        .inputs(inputs)
        .build();

    try {
        ResponseEntity response = restTemplate.postForEntity(
            triggerUrl,
            executionRequest,
            KestraExecutionResponse.class
        );

        if (response.getStatusCode() == HttpStatus.OK) {
            log.info("Kestra workflow triggered successfully. Execution ID: {}",
                response.getBody().getExecutionId());
        }
    } catch (Exception e) {
        log.error("Failed to trigger Kestra workflow", e);
        throw new BookingProcessingException("Failed to initiate booking workflow", e);
    }
}

@Data
@Builder
class KestraExecutionRequest {
    private String namespace;
    private String flowId;
    private Map inputs;
}

@Data
class KestraExecutionResponse {
    private String executionId;
    private String state;
}

```

Here's how the integration works:

1. **Booking Service Integration**:
 - The `BookingService` creates a booking record in the database
 - Prepares the necessary data for the Kestra workflow
 - Triggers the workflow using Kestra's REST API
 - Handles any errors that occur during the process
2. **Kestra Workflow**:
 - Receives booking details as input
 - Validates the booking
 - Checks resource availability
 - Sends confirmation emails
 - Updates booking status
 - Handles errors with notifications
3. **Key Features**:
 - Asynchronous processing
 - Error handling and notifications
 - Database integration
 - Email notifications
 - Status tracking
4. **Configuration Required**:

```
``properties
  kestra.api.url=http://kestra-server:8080
``
```

5. ****Security Considerations****:

- Use API keys or tokens for Kestra API authentication
- Secure database credentials using secrets management
- Implement proper error handling and logging
- Use HTTPS for API communications

To implement this in your system:

1. Add the Kestra client dependencies to your project
2. Configure the Kestra API endpoint
3. Deploy the workflow YAML to your Kestra server
4. Implement error handling and monitoring
5. Set up the required database connections and email configurations

Would you like me to:

1. Add more detailed error handling?
2. Show how to implement specific tasks in the workflow?
3. Add monitoring and tracking capabilities?
4. Demonstrate how to handle specific booking scenarios?

USER

i need to implement it in the nestjs

BOT

> **Routing to Claude Sonnet 3.5**

```
// src/booking/booking.module.ts
import { Module, HttpModule } from '@nestjs/common';
import { BookingService } from './booking.service';
import { BookingController } from './booking.controller';
import { KestraService } from './kestra/kestra.service';
import { TypeOrmModule } from '@nestjs/typeorm';
import { Booking } from './entities/booking.entity';

@Module({
  imports: [
    HttpModule,
    TypeOrmModule.forFeature([Booking]),
  ],
  controllers: [BookingController],
  providers: [BookingService, KestraService],
})
export class BookingModule {}

// src/kestra/kestra.service.ts
import { Injectable, HttpService } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
import { KestraExecutionRequest } from './interfaces/kestra-execution.interface';
```

```

@Injectable()
export class KestraService {
  constructor(
    private readonly httpService: HttpService,
    private readonly configService: ConfigService,
  ) {}

  private readonly kestraApiUrl = this.configService.get('KESTRA_API_URL');

  async triggerWorkflow(workflowId: string, inputs: Record) {
    const executionRequest: KestraExecutionRequest = {
      namespace: 'logistics',
      flowId: workflowId,
      inputs,
    };

    try {
      const response = await this.httpService
        .post(`${this.kestraApiUrl}/api/v1/executions/trigger`, executionRequest)
        .toPromise();

      return response.data;
    } catch (error) {
      throw new Error(`Failed to trigger Kestra workflow: ${error.message}`);
    }
  }

  async getWorkflowStatus(executionId: string) {
    try {
      const response = await this.httpService
        .get(`${this.kestraApiUrl}/api/v1/executions/${executionId}`)
        .toPromise();

      return response.data;
    } catch (error) {
      throw new Error(`Failed to get workflow status: ${error.message}`);
    }
  }
}

```

```

// src/booking/booking.service.ts
import { Injectable, Logger } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { Booking } from './entities/booking.entity';
import { CreateBookingDto } from './dto/create-booking.dto';
import { KestraService } from '../kestra/kestra.service';

```

```

@Injectable()
export class BookingService {
  private readonly logger = new Logger(BookingService.name);

  constructor(
    @InjectRepository(Booking)
    private readonly bookingRepository: Repository,
    private readonly kestraService: KestraService,
  ) {}

  async createBooking(createBookingDto: CreateBookingDto) {
    // 1. Create booking entity
    const booking = this.bookingRepository.create(createBookingDto);
    await this.bookingRepository.save(booking);

    // 2. Prepare workflow input
    const workflowInput = {
      bookingId: booking.id,
      customerId: booking.customerId,
      deliveryAddress: booking.deliveryAddress,
    };
  }
}

```



```

    scheduledDate: booking.scheduledDate,
    items: booking.items,
  };

  try {
    // 3. Trigger Kestra workflow
    const workflowResponse = await this.kestraService.triggerWorkflow(
      'logistics-booking-flow',
      workflowInput,
    );

    // 4. Update booking with workflow execution ID
    booking.workflowExecutionId = workflowResponse.executionId;
    await this.bookingRepository.save(booking);

    return {
      booking,
      workflowExecutionId: workflowResponse.executionId,
    };
  } catch (error) {
    this.logger.error(`Failed to process booking: ${error.message}`, error.stack);
    throw error;
  }
}

async getBookingStatus(bookingId: string) {
  const booking = await this.bookingRepository.findOne(bookingId);
  if (!booking) {
    throw new Error('Booking not found');
  }

  if (booking.workflowExecutionId) {
    const workflowStatus = await this.kestraService.getWorkflowStatus(
      booking.workflowExecutionId,
    );
    return {
      booking,
      workflowStatus,
    };
  }

  return { booking };
}
}

// src/booking/entities/booking.entity.ts
import { Entity, Column, PrimaryGeneratedColumn } from 'typeorm';

@Entity()
export class Booking {
  @PrimaryGeneratedColumn('uuid')
  id: string;

  @Column()
  customerId: string;

  @Column()
  deliveryAddress: string;

  @Column()
  scheduledDate: Date;

  @Column('json')
  items: any[];

  @Column({ nullable: true })
  workflowExecutionId: string;

```

```

@Column({
  type: 'enum',
  enum: ['PENDING', 'PROCESSING', 'COMPLETED', 'FAILED'],
  default: 'PENDING',
})
status: string;
}

// src/booking/dto/create-booking.dto.ts
import { IsNotEmpty, IsString, IsDate, IsArray } from 'class-validator';

export class CreateBookingDto {
  @IsNotEmpty()
  @IsString()
  customerId: string;

  @IsNotEmpty()
  @IsString()
  deliveryAddress: string;

  @IsNotEmpty()
  @IsDate()
  scheduledDate: Date;

  @IsNotEmpty()
  @IsArray()
  items: any[];
}

// src/kestra/interfaces/kestra-execution.interface.ts
export interface KestraExecutionRequest {
  namespace: string;
  flowId: string;
  inputs: Record;
}

export interface KestraExecutionResponse {
  executionId: string;
  state: string;
}

```

To complete the implementation, you'll also need to create a Kestra workflow YAML file. Here's an example:

```

typescript
// src/booking/booking.module.ts
import { Module, HttpModule } from '@nestjs/common';
import { BookingService } from './booking.service';
import { BookingController } from './booking.controller';
import { KestraService } from '../kestra/kestra.service';
import { TypeOrmModule } from '@nestjs/typeorm';
import { Booking } from './entities/booking.entity';

@Module({
  imports: [
    HttpModule,
    TypeOrmModule.forFeature([Booking]),
  ],
  controllers: [BookingController],
  providers: [BookingService, KestraService],
})
export class BookingModule {}

// src/kestra/kestra.service.ts
import { Injectable, HttpService } from '@nestjs/common';

```

```
import { ConfigService } from '@nestjs/config';
import { KestraExecutionRequest } from './interfaces/kestra-execution.interface';
```

```
@Injectable()
export class KestraService {
  constructor(
    private readonly httpService: HttpService,
    private readonly configService: ConfigService,
  ) {}

  private readonly kestraApiUrl = this.configService.get('KESTRA_API_URL');

  async triggerWorkflow(workflowId: string, inputs: Record) {
    const executionRequest: KestraExecutionRequest = {
      namespace: 'logistics',
      flowId: workflowId,
      inputs,
    };

    try {
      const response = await this.httpService
        .post(`${this.kestraApiUrl}/api/v1/executions/trigger`, executionRequest)
        .toPromise();

      return response.data;
    } catch (error) {
      throw new Error(`Failed to trigger Kestra workflow: ${error.message}`);
    }
  }
}
```

```
async getWorkflowStatus(executionId: string) {
  try {
    const response = await this.httpService
      .get(`${this.kestraApiUrl}/api/v1/executions/${executionId}`)
      .toPromise();

    return response.data;
  } catch (error) {
    throw new Error(`Failed to get workflow status: ${error.message}`);
  }
}
```

```
// src/booking/booking.service.ts
import { Injectable, Logger } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { Booking } from './entities/booking.entity';
import { CreateBookingDto } from './dto/create-booking.dto';
import { KestraService } from '../kestra/kestra.service';
```

```
@Injectable()
export class BookingService {
  private readonly logger = new Logger(BookingService.name);

  constructor(
    @InjectRepository(Booking)
    private readonly bookingRepository: Repository,
    private readonly kestraService: KestraService,
  ) {}

  async createBooking(createBookingDto: CreateBookingDto) {
    // 1. Create booking entity
    const booking = this.bookingRepository.create(createBookingDto);
    await this.bookingRepository.save(booking);

    // 2. Prepare workflow input
    const workflowInput = {
```

```

    bookingId: booking.id,
    customerId: booking.customerId,
    deliveryAddress: booking.deliveryAddress,
    scheduledDate: booking.scheduledDate,
    items: booking.items,
  };

  try {
    // 3. Trigger Kestra workflow
    const workflowResponse = await this.kestraService.triggerWorkflow(
      'logistics-booking-flow',
      workflowInput,
    );

    // 4. Update booking with workflow execution ID
    booking.workflowExecutionId = workflowResponse.executionId;
    await this.bookingRepository.save(booking);

    return {
      booking,
      workflowExecutionId: workflowResponse.executionId,
    };
  } catch (error) {
    this.logger.error(`Failed to process booking: ${error.message}`, error.stack);
    throw error;
  }
}

async getBookingStatus(bookingId: string) {
  const booking = await this.bookingRepository.findOne(bookingId);
  if (!booking) {
    throw new Error('Booking not found');
  }

  if (booking.workflowExecutionId) {
    const workflowStatus = await this.kestraService.getWorkflowStatus(
      booking.workflowExecutionId,
    );
    return {
      booking,
      workflowStatus,
    };
  }

  return { booking };
}
}

// src/booking/entities/booking.entity.ts
import { Entity, Column, PrimaryGeneratedColumn } from 'typeorm';

@Entity()
export class Booking {
  @PrimaryGeneratedColumn('uuid')
  id: string;

  @Column()
  customerId: string;

  @Column()
  deliveryAddress: string;

  @Column()
  scheduledDate: Date;

  @Column('json')
  items: any[];
}

```

```

@Column({ nullable: true })
workflowExecutionId: string;

@Column({
  type: 'enum',
  enum: ['PENDING', 'PROCESSING', 'COMPLETED', 'FAILED'],
  default: 'PENDING',
})
status: string;
}

// src/booking/dto/create-booking.dto.ts
import { IsNotEmpty, IsString, IsDate, IsArray } from 'class-validator';

export class CreateBookingDto {
  @IsNotEmpty()
  @IsString()
  customerId: string;

  @IsNotEmpty()
  @IsString()
  deliveryAddress: string;

  @IsNotEmpty()
  @IsDate()
  scheduledDate: Date;

  @IsNotEmpty()
  @IsArray()
  items: any[];
}

// src/kestra/interfaces/kestra-execution.interface.ts
export interface KestraExecutionRequest {
  namespace: string;
  flowId: string;
  inputs: Record;
}

export interface KestraExecutionResponse {
  executionId: string;
  state: string;
}

```

To use this implementation:

1. ****Installation****:

```

```bash
npm install @nestjs/axios @nestjs/config class-validator class-transformer
```

```

2. ****Environment Configuration****:

Create a `.env`` file:

```

```env
KESTRA_API_URL=http://your-kestra-server:8080
```

```

3. **Usage Example**:

```
``typescript
// In your controller
@Post()
async createBooking(@Body() createBookingDto: CreateBookingDto) {
  return this.bookingService.createBooking(createBookingDto);
}

@Get('/:id/status')
async getBookingStatus(@Param('id') id: string) {
  return this.bookingService.getBookingStatus(id);
}
``
```

Key Features:

1. **Modular Architecture**: Separate services for booking and Kestra integration
2. **Type Safety**: TypeScript interfaces and DTOs
3. **Error Handling**: Comprehensive error handling and logging
4. **Status Tracking**: Workflow execution tracking
5. **Database Integration**: TypeORM entity and repository
6. **Validation**: DTO validation using class-validator

Would you like me to:

1. Add more error handling scenarios?
2. Implement additional Kestra workflow tasks?
3. Add monitoring and metrics collection?
4. Show how to implement webhook handlers for Kestra callbacks?

USER

give all the code for it

BOT

> Routing to Claude Sonnet 3.5

1. First, let's set up the project structure:

```
# Create new NestJS project
nest new logistics-booking-system
cd logistics-booking-system

# Install required dependencies
npm install @nestjs/axios @nestjs/config @nestjs/typeorm typeorm pg class-validator class-transformer
```

```
``
```

2. Main Application Module:

```
'''  
  
# Create new NestJS project  
nest new logistics-booking-system  
cd logistics-booking-system  
  
# Install required dependencies  
npm install @nestjs/axios @nestjs/config @nestjs/typeorm typeorm pg class-validator class-transformer
```

3. Kestra Module and Service:

```
# Create new NestJS project  
nest new logistics-booking-system  
cd logistics-booking-system  
  
# Install required dependencies  
npm install @nestjs/axios @nestjs/config @nestjs/typeorm typeorm pg class-validator class-transformer
```

4. Booking Module Implementation:

```
# Create new NestJS project  
nest new logistics-booking-system  
cd logistics-booking-system  
  
# Install required dependencies  
npm install @nestjs/axios @nestjs/config @nestjs/typeorm typeorm pg class-validator class-transformer
```

'''

5. Kestra Workflow Definition:

'''

```
# Create new NestJS project  
nest new logistics-booking-system  
cd logistics-booking-system  
  
# Install required dependencies  
npm install @nestjs/axios @nestjs/config @nestjs/typeorm typeorm pg class-validator class-transformer
```

6. Environment Configuration:

```
# Create new NestJS project  
nest new logistics-booking-system  
cd logistics-booking-system  
  
# Install required dependencies
```

```
npm install @nestjs/axios @nestjs/config @nestjs/typeorm typeorm pg class-validator class-transformer
```

To run the application:

1. Start your PostgreSQL database
2. Start your Kestra server
3. Run the NestJS application:

```
```bash
npm run start:dev
```
```

Test the API:

```
```bash
Create a booking
curl -X POST http://localhost:3000/bookings \
-H "Content-Type: application/json" \
-d '{
 "customerId": "customer123",
 "deliveryAddress": "123 Main St",
 "scheduledDate": "2024-01-01T10:00:00Z",
 "items": [
 {
 "itemId": "item123",
 "quantity": 2
 }
]
}'

Check booking status
curl http://localhost:3000/bookings/{booking-id}/status
```
```

Would you like me to:

1. Add more error handling scenarios?
2. Implement additional Kestra workflow tasks?
3. Add monitoring and metrics collection?
4. Show how to implement webhook handlers for Kestra callbacks?
5. Add authentication and authorization?

USER

json print the code i am not able to open it

BOT

> Routing to Claude Sonnet 3.5

Let me provide the complete code structure in a clear format:


```

// src/app.module.ts
import { Module } from '@nestjs/common';
import { ConfigModule } from '@nestjs/config';
import { TypeOrmModule } from '@nestjs/typeorm';
import { BookingModule } from './booking/booking.module';
import { KestraModule } from './kestra/kestra.module';

@Module({
  imports: [
    ConfigModule.forRoot({
      isGlobal: true,
    }),
    TypeOrmModule.forRoot({
      type: 'postgres',
      host: process.env.DB_HOST,
      port: parseInt(process.env.DB_PORT),
      username: process.env.DB_USERNAME,
      password: process.env.DB_PASSWORD,
      database: process.env.DB_NAME,
      entities: [__dirname + '**/*.entity{.ts,.js}'],
      synchronize: true, // set to false in production
    }),
    BookingModule,
    KestraModule,
  ],
})
export class AppModule {}

// src/kestra/kestra.module.ts
import { Module, HttpModule } from '@nestjs/common';
import { KestraService } from './kestra.service';

@Module({
  imports: [HttpModule],
  providers: [KestraService],
  exports: [KestraService],
})
export class KestraModule {}

// src/kestra/kestra.service.ts
import { Injectable, HttpService, Logger } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
import { KestraExecutionRequest, KestraExecutionResponse } from './interfaces/kestra.interfaces';

@Injectable()
export class KestraService {
  private readonly logger = new Logger(KestraService.name);

  constructor(
    private readonly httpService: HttpService,
    private readonly configService: ConfigService,
  ) {}

  async triggerWorkflow(workflowId: string, inputs: Record): Promise {
    const kestraApiUrl = this.configService.get('KESTRA_API_URL');
    const request: KestraExecutionRequest = {
      namespace: 'logistics',
      flowId: workflowId,
      inputs,
    };

    try {
      const response = await this.httpService
        .post(`${kestraApiUrl}/api/v1/executions/trigger`, request)
        .toPromise();

      return response.data;
    }
  }
}

```

```

    } catch (error) {
      this.logger.error(`Failed to trigger Kestra workflow: ${error.message}`);
      throw new Error(`Workflow trigger failed: ${error.message}`);
    }
  }

  async getWorkflowStatus(executionId: string): Promise {
    const kestraApiUrl = this.configService.get('KESTRA_API_URL');

    try {
      const response = await this.httpService
        .get(`${kestraApiUrl}/api/v1/executions/${executionId}`)
        .toPromise();

      return response.data;
    } catch (error) {
      this.logger.error(`Failed to get workflow status: ${error.message}`);
      throw new Error(`Status check failed: ${error.message}`);
    }
  }
}

// src/booking/booking.module.ts
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { BookingController } from './booking.controller';
import { BookingService } from './booking.service';
import { Booking } from './entities/booking.entity';
import { KestraModule } from '../kestra/kestra.module';

@Module({
  imports: [
    TypeOrmModule.forFeature([Booking]),
    KestraModule,
  ],
  controllers: [BookingController],
  providers: [BookingService],
})
export class BookingModule {}

// src/booking/booking.controller.ts
import { Controller, Post, Body, Get, Param, HttpException, HttpStatus } from '@nestjs/common';
import { BookingService } from './booking.service';
import { CreateBookingDto } from '../dto/create-booking.dto';

@Controller('bookings')
export class BookingController {
  constructor(private readonly bookingService: BookingService) {}

  @Post()
  async createBooking(@Body() createBookingDto: CreateBookingDto) {
    try {
      return await this.bookingService.createBooking(createBookingDto);
    } catch (error) {
      throw new HttpException(error.message, HttpStatus.BAD_REQUEST);
    }
  }

  @Get('/:id/status')
  async getBookingStatus(@Param('id') id: string) {
    try {
      return await this.bookingService.getBookingStatus(id);
    } catch (error) {
      throw new HttpException(error.message, HttpStatus.NOT_FOUND);
    }
  }
}

```

```

// src/booking/booking.service.ts
import { Injectable, Logger } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { Booking } from '../entities/booking.entity';
import { CreateBookingDto } from '../dto/create-booking.dto';
import { KestraService } from '../kestra/kestra.service';

@Injectable()
export class BookingService {
  private readonly logger = new Logger(BookingService.name);

  constructor(
    @InjectRepository(Booking)
    private readonly bookingRepository: Repository,
    private readonly kestraService: KestraService,
  ) {}

  async createBooking(createBookingDto: CreateBookingDto) {
    // Create booking record
    const booking = this.bookingRepository.create({
      ...createBookingDto,
      status: 'PENDING',
    });
    await this.bookingRepository.save(booking);

    try {
      // Trigger Kestra workflow
      const workflowResponse = await this.kestraService.triggerWorkflow(
        'logistics-booking-flow',
        {
          bookingId: booking.id,
          customerId: booking.customerId,
          deliveryAddress: booking.deliveryAddress,
          scheduledDate: booking.scheduledDate,
          items: booking.items,
        },
      );

      // Update booking with workflow execution ID
      booking.workflowExecutionId = workflowResponse.executionId;
      booking.status = 'PROCESSING';
      await this.bookingRepository.save(booking);

      return {
        booking,
        workflowExecutionId: workflowResponse.executionId,
      };
    } catch (error) {
      booking.status = 'FAILED';
      await this.bookingRepository.save(booking);
      throw error;
    }
  }

  async getBookingStatus(id: string) {
    const booking = await this.bookingRepository.findOne({ where: { id } });
    if (!booking) {
      throw new Error('Booking not found');
    }

    if (booking.workflowExecutionId) {
      const workflowStatus = await this.kestraService.getWorkflowStatus(
        booking.workflowExecutionId,
      );
      return {
        booking,
        workflowStatus,
      };
    }
  }
}

```

```

    };
  }

  return { booking };
}
}

// src/booking/entities/booking.entity.ts
import { Entity, Column, PrimaryGeneratedColumn, CreateDateColumn, UpdateDateColumn } from 'typeorm';

@Entity()
export class Booking {
  @PrimaryGeneratedColumn('uuid')
  id: string;

  @Column()
  customerId: string;

  @Column()
  deliveryAddress: string;

  @Column()
  scheduledDate: Date;

  @Column('jsonb')
  items: any[];

  @Column({ nullable: true })
  workflowExecutionId: string;

  @Column({
    type: 'enum',
    enum: ['PENDING', 'PROCESSING', 'COMPLETED', 'FAILED'],
    default: 'PENDING',
  })
  status: string;

  @CreateDateColumn()
  createdAt: Date;

  @UpdateDateColumn()
  updatedAt: Date;
}

// src/booking/dto/create-booking.dto.ts
import { IsNotEmpty, IsString, IsDate, IsArray, ValidateNested } from 'class-validator';
import { Type } from 'class-transformer';

export class BookingItemDto {
  @IsString()
  @IsNotEmpty()
  itemId: string;

  @IsNotEmpty()
  quantity: number;
}

export class CreateBookingDto {
  @IsNotEmpty()
  @IsString()
  customerId: string;

  @IsNotEmpty()
  @IsString()
  deliveryAddress: string;

  @IsNotEmpty()
  @Type(() => Date)

```

```

@IsDate()
scheduledDate: Date;

@IsArray()
@ValidateNested({ each: true })
@Type(() => BookingItemDto)
items: BookingItemDto[];
}

// src/kestra/interfaces/kestra.interfaces.ts
export interface KestraExecutionRequest {
  namespace: string;
  flowId: string;
  inputs: Record;
}

export interface KestraExecutionResponse {
  executionId: string;
  state: string;
}

```

And here's the environment configuration:

```

typescript
// src/app.module.ts
import { Module } from '@nestjs/common';
import { ConfigModule } from '@nestjs/config';
import { TypeOrmModule } from '@nestjs/typeorm';
import { BookingModule } from '../booking/booking.module';
import { KestraModule } from '../kestra/kestra.module';

@Module({
  imports: [
    ConfigModule.forRoot({
      isGlobal: true,
    }),
    TypeOrmModule.forRoot({
      type: 'postgres',
      host: process.env.DB_HOST,
      port: parseInt(process.env.DB_PORT),
      username: process.env.DB_USERNAME,
      password: process.env.DB_PASSWORD,
      database: process.env.DB_NAME,
      entities: [__dirname + '**/*.entity{.ts,.js}'],
      synchronize: true, // set to false in production
    }),
    BookingModule,
    KestraModule,
  ],
})
export class AppModule {}

// src/kestra/kestra.module.ts
import { Module, HttpModule } from '@nestjs/common';
import { KestraService } from '../kestra.service';

@Module({
  imports: [HttpModule],
  providers: [KestraService],
  exports: [KestraService],
})
export class KestraModule {}

// src/kestra/kestra.service.ts

```

```
import { Injectable, HttpService, Logger } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
import { KestraExecutionRequest, KestraExecutionResponse } from './interfaces/kestra.interfaces';
```

```
@Injectable()
export class KestraService {
  private readonly logger = new Logger(KestraService.name);

  constructor(
    private readonly httpService: HttpService,
    private readonly configService: ConfigService,
  ) {}

  async triggerWorkflow(workflowId: string, inputs: Record): Promise {
    const kestraApiUrl = this.configService.get('KESTRA_API_URL');
    const request: KestraExecutionRequest = {
      namespace: 'logistics',
      flowId: workflowId,
      inputs,
    };

    try {
      const response = await this.httpService
        .post(`${kestraApiUrl}/api/v1/executions/trigger`, request)
        .toPromise();

      return response.data;
    } catch (error) {
      this.logger.error(`Failed to trigger Kestra workflow: ${error.message}`);
      throw new Error(`Workflow trigger failed: ${error.message}`);
    }
  }

  async getWorkflowStatus(executionId: string): Promise {
    const kestraApiUrl = this.configService.get('KESTRA_API_URL');

    try {
      const response = await this.httpService
        .get(`${kestraApiUrl}/api/v1/executions/${executionId}`)
        .toPromise();

      return response.data;
    } catch (error) {
      this.logger.error(`Failed to get workflow status: ${error.message}`);
      throw new Error(`Status check failed: ${error.message}`);
    }
  }
}
```

```
// src/booking/booking.module.ts
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { BookingController } from './booking.controller';
import { BookingService } from './booking.service';
import { Booking } from './entities/booking.entity';
import { KestraModule } from '../kestra/kestra.module';
```

```
@Module({
  imports: [
    TypeOrmModule.forFeature([Booking]),
    KestraModule,
  ],
  controllers: [BookingController],
  providers: [BookingService],
})
export class BookingModule {}
```

```
// src/booking/booking.controller.ts
```

```
import { Controller, Post, Body, Get, Param, HttpException, HttpStatus } from '@nestjs/common';
import { BookingService } from '../booking.service';
import { CreateBookingDto } from '../dto/create-booking.dto';
```

```
@Controller('bookings')
export class BookingController {
  constructor(private readonly bookingService: BookingService) {}

  @Post()
  async createBooking(@Body() createBookingDto: CreateBookingDto) {
    try {
      return await this.bookingService.createBooking(createBookingDto);
    } catch (error) {
      throw new HttpException(error.message, HttpStatus.BAD_REQUEST);
    }
  }

  @Get('/:id/status')
  async getBookingStatus(@Param('id') id: string) {
    try {
      return await this.bookingService.getBookingStatus(id);
    } catch (error) {
      throw new HttpException(error.message, HttpStatus.NOT_FOUND);
    }
  }
}
```

```
// src/booking/booking.service.ts
import { Injectable, Logger } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { Booking } from '../entities/booking.entity';
import { CreateBookingDto } from '../dto/create-booking.dto';
import { KestraService } from '../kestra/kestra.service';
```

```
@Injectable()
export class BookingService {
  private readonly logger = new Logger(BookingService.name);

  constructor(
    @InjectRepository(Booking)
    private readonly bookingRepository: Repository,
    private readonly kestraService: KestraService,
  ) {}

  async createBooking(createBookingDto: CreateBookingDto) {
    // Create booking record
    const booking = this.bookingRepository.create({
      ...createBookingDto,
      status: 'PENDING',
    });
    await this.bookingRepository.save(booking);

    try {
      // Trigger Kestra workflow
      const workflowResponse = await this.kestraService.triggerWorkflow(
        'logistics-booking-flow',
        {
          bookingId: booking.id,
          customerId: booking.customerId,
          deliveryAddress: booking.deliveryAddress,
          scheduledDate: booking.scheduledDate,
          items: booking.items,
        },
      );

      // Update booking with workflow execution ID
      booking.workflowExecutionId = workflowResponse.executionId;
    } catch (error) {
      // Handle error
    }
  }
}
```

```

        booking.status = 'PROCESSING';
        await this.bookingRepository.save(booking);

        return {
            booking,
            workflowExecutionId: workflowResponse.executionId,
        };
    } catch (error) {
        booking.status = 'FAILED';
        await this.bookingRepository.save(booking);
        throw error;
    }
}

async getBookingStatus(id: string) {
    const booking = await this.bookingRepository.findOne({ where: { id } });
    if (!booking) {
        throw new Error('Booking not found');
    }

    if (booking.workflowExecutionId) {
        const workflowStatus = await this.kestraService.getWorkflowStatus(
            booking.workflowExecutionId,
        );
        return {
            booking,
            workflowStatus,
        };
    }

    return { booking };
}
}

// src/booking/entities/booking.entity.ts
import { Entity, Column, PrimaryGeneratedColumn, CreateDateColumn, UpdateDateColumn } from 'typeorm';

@Entity()
export class Booking {
    @PrimaryGeneratedColumn('uuid')
    id: string;

    @Column()
    customerId: string;

    @Column()
    deliveryAddress: string;

    @Column()
    scheduledDate: Date;

    @Column('jsonb')
    items: any[];

    @Column({ nullable: true })
    workflowExecutionId: string;

    @Column({
        type: 'enum',
        enum: ['PENDING', 'PROCESSING', 'COMPLETED', 'FAILED'],
        default: 'PENDING',
    })
    status: string;

    @CreateDateColumn()
    createdAt: Date;

    @UpdateDateColumn()

```



```

    updatedAt: Date;
  }

// src/booking/dto/create-booking.dto.ts
import { IsNotEmpty, IsString, IsDate, IsArray, ValidateNested } from 'class-validator';
import { Type } from 'class-transformer';

export class BookingItemDto {
  @IsString()
  @IsNotEmpty()
  itemId: string;

  @IsNotEmpty()
  quantity: number;
}

export class CreateBookingDto {
  @IsNotEmpty()
  @IsString()
  customerId: string;

  @IsNotEmpty()
  @IsString()
  deliveryAddress: string;

  @IsNotEmpty()
  @Type(() => Date)
  @IsDate()
  scheduledDate: Date;

  @IsArray()
  @ValidateNested({ each: true })
  @Type(() => BookingItemDto)
  items: BookingItemDto[];
}

// src/kestra/interfaces/kestra.interfaces.ts
export interface KestraExecutionRequest {
  namespace: string;
  flowId: string;
  inputs: Record;
}

export interface KestraExecutionResponse {
  executionId: string;
  state: string;
}

```

080

...

And the Kestra workflow YAML:

...

```

typescript
// src/app.module.ts
import { Module } from '@nestjs/common';
import { ConfigModule } from '@nestjs/config';
import { TypeOrmModule } from '@nestjs/typeorm';
import { BookingModule } from '../booking/booking.module';
import { KestraModule } from '../kestra/kestra.module';

@Module({

```

```

imports: [
  ConfigModule.forRoot({
    isGlobal: true,
  }),
  TypeOrmModule.forRoot({
    type: 'postgres',
    host: process.env.DB_HOST,
    port: parseInt(process.env.DB_PORT),
    username: process.env.DB_USERNAME,
    password: process.env.DB_PASSWORD,
    database: process.env.DB_NAME,
    entities: [__dirname + '/**/*.entity{.ts,.js}'],
    synchronize: true, // set to false in production
  }),
  BookingModule,
  KestraModule,
],
})
export class AppModule {}

// src/kestra/kestra.module.ts
import { Module, HttpModule } from '@nestjs/common';
import { KestraService } from './kestra.service';

@Module({
  imports: [HttpModule],
  providers: [KestraService],
  exports: [KestraService],
})
export class KestraModule {}

// src/kestra/kestra.service.ts
import { Injectable, HttpService, Logger } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
import { KestraExecutionRequest, KestraExecutionResponse } from './interfaces/kestra.interfaces';

@Injectable()
export class KestraService {
  private readonly logger = new Logger(KestraService.name);

  constructor(
    private readonly httpService: HttpService,
    private readonly configService: ConfigService,
  ) {}

  async triggerWorkflow(workflowId: string, inputs: Record): Promise {
    const kestraApiUrl = this.configService.get('KESTRA_API_URL');
    const request: KestraExecutionRequest = {
      namespace: 'logistics',
      flowId: workflowId,
      inputs,
    };

    try {
      const response = await this.httpService
        .post(`${kestraApiUrl}/api/v1/executions/trigger`, request)
        .toPromise();

      return response.data;
    } catch (error) {
      this.logger.error(`Failed to trigger Kestra workflow: ${error.message}`);
      throw new Error(`Workflow trigger failed: ${error.message}`);
    }
  }

  async getWorkflowStatus(executionId: string): Promise {
    const kestraApiUrl = this.configService.get('KESTRA_API_URL');

```

```

    try {
      const response = await this.httpService
        .get(`${kestraApiUrl}/api/v1/executions/${executionId}`)
        .toPromise();

      return response.data;
    } catch (error) {
      this.logger.error(`Failed to get workflow status: ${error.message}`);
      throw new Error(`Status check failed: ${error.message}`);
    }
  }
}

// src/booking/booking.module.ts
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { BookingController } from './booking.controller';
import { BookingService } from './booking.service';
import { Booking } from './entities/booking.entity';
import { KestraModule } from '../kestra/kestra.module';

@Module({
  imports: [
    TypeOrmModule.forFeature([Booking]),
    KestraModule,
  ],
  controllers: [BookingController],
  providers: [BookingService],
})
export class BookingModule {}

// src/booking/booking.controller.ts
import { Controller, Post, Body, Get, Param, HttpException, HttpStatus } from '@nestjs/common';
import { BookingService } from './booking.service';
import { CreateBookingDto } from './dto/create-booking.dto';

@Controller('bookings')
export class BookingController {
  constructor(private readonly bookingService: BookingService) {}

  @Post()
  async createBooking(@Body() createBookingDto: CreateBookingDto) {
    try {
      return await this.bookingService.createBooking(createBookingDto);
    } catch (error) {
      throw new HttpException(error.message, HttpStatus.BAD_REQUEST);
    }
  }

  @Get('/:id/status')
  async getBookingStatus(@Param('id') id: string) {
    try {
      return await this.bookingService.getBookingStatus(id);
    } catch (error) {
      throw new HttpException(error.message, HttpStatus.NOT_FOUND);
    }
  }
}

// src/booking/booking.service.ts
import { Injectable, Logger } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { Booking } from './entities/booking.entity';
import { CreateBookingDto } from './dto/create-booking.dto';
import { KestraService } from '../kestra/kestra.service';

@Injectable()

```

```

export class BookingService {
  private readonly logger = new Logger(BookingService.name);

  constructor(
    @InjectRepository(Booking)
    private readonly bookingRepository: Repository,
    private readonly kestraService: KestraService,
  ) {}

  async createBooking(createBookingDto: CreateBookingDto) {
    // Create booking record
    const booking = this.bookingRepository.create({
      ...createBookingDto,
      status: 'PENDING',
    });
    await this.bookingRepository.save(booking);

    try {
      // Trigger Kestra workflow
      const workflowResponse = await this.kestraService.triggerWorkflow(
        'logistics-booking-flow',
        {
          bookingId: booking.id,
          customerId: booking.customerId,
          deliveryAddress: booking.deliveryAddress,
          scheduledDate: booking.scheduledDate,
          items: booking.items,
        },
      );

      // Update booking with workflow execution ID
      booking.workflowExecutionId = workflowResponse.executionId;
      booking.status = 'PROCESSING';
      await this.bookingRepository.save(booking);

      return {
        booking,
        workflowExecutionId: workflowResponse.executionId,
      };
    } catch (error) {
      booking.status = 'FAILED';
      await this.bookingRepository.save(booking);
      throw error;
    }
  }

  async getBookingStatus(id: string) {
    const booking = await this.bookingRepository.findOne({ where: { id } });
    if (!booking) {
      throw new Error('Booking not found');
    }

    if (booking.workflowExecutionId) {
      const workflowStatus = await this.kestraService.getWorkflowStatus(
        booking.workflowExecutionId,
      );
      return {
        booking,
        workflowStatus,
      };
    }

    return { booking };
  }
}

// src/booking/entities/booking.entity.ts
import { Entity, Column, PrimaryGeneratedColumn, CreateDateColumn, UpdateDateColumn } from 'typeorm';

```

```

@Entity()
export class Booking {
  @PrimaryGeneratedColumn('uuid')
  id: string;

  @Column()
  customerId: string;

  @Column()
  deliveryAddress: string;

  @Column()
  scheduledDate: Date;

  @Column('jsonb')
  items: any[];

  @Column({ nullable: true })
  workflowExecutionId: string;

  @Column({
    type: 'enum',
    enum: ['PENDING', 'PROCESSING', 'COMPLETED', 'FAILED'],
    default: 'PENDING',
  })
  status: string;

  @CreateDateColumn()
  createdAt: Date;

  @UpdateDateColumn()
  updatedAt: Date;
}

// src/booking/dto/create-booking.dto.ts
import { IsNotEmpty, IsString, IsDate, IsArray, ValidateNested } from 'class-validator';
import { Type } from 'class-transformer';

export class BookingItemDto {
  @IsString()
  @IsNotEmpty()
  itemId: string;

  @IsNotEmpty()
  quantity: number;
}

export class CreateBookingDto {
  @IsNotEmpty()
  @IsString()
  customerId: string;

  @IsNotEmpty()
  @IsString()
  deliveryAddress: string;

  @IsNotEmpty()
  @Type(() => Date)
  @IsDate()
  scheduledDate: Date;

  @IsArray()
  @ValidateNested({ each: true })
  @Type(() => BookingItemDto)
  items: BookingItemDto[];
}

```

```
// src/kestra/interfaces/kestra.interfaces.ts
export interface KestraExecutionRequest {
  namespace: string;
  flowId: string;
  inputs: Record;
}

export interface KestraExecutionResponse {
  executionId: string;
  state: string;
}
```

To use this implementation:

1. Create a new NestJS project and install dependencies:

```
```bash
nest new logistics-booking-system
cd logistics-booking-system
npm install @nestjs/axios @nestjs/config @nestjs/typeorm typeorm pg class-validator class-transformer
```
```

2. Copy all the files into their respective directories following the NestJS structure:

```
...
src/
├── app.module.ts
├── kestra/
│   ├── kestra.module.ts
│   ├── kestra.service.ts
│   └── interfaces/
│       └── kestra.interfaces.ts
├── booking/
│   ├── booking.module.ts
│   ├── booking.controller.ts
│   ├── booking.service.ts
│   ├── entities/
│   │   └── booking.entity.ts
│   └── dto/
│       └── create-booking.dto.ts
...

```

3. Create `.env` file in the root directory with the provided configuration

4. Deploy the Kestra workflow YAML to your Kestra server

5. Start the application:

```
```bash
npm run start:dev
```
```

Would you like me to:

1. Add authentication middleware?
2. Implement webhook handlers for Kestra callbacks?

3. Add more error handling scenarios?
4. Add monitoring and metrics collection?
5. Implement additional booking features?