Toplevel-Documentation.md

Project Overview:

The Spring PetClinic Microservices application is a cloud-native version of the classic Spring PetClinic demo application. It demonstrates how to build a distributed system using Spring Cloud and microservices architecture principles. The application allows pet clinic staff to manage pet owners, their pets, veterinarians, and visit records through a web-based interface.

The system is decomposed into several microservices, each responsible for a specific business capability. These services communicate with each other through REST APIs and are designed to be independently deployable, scalable, and resilient. The architecture leverages modern cloud-native patterns such as service discovery, centralized configuration, API gateway, and circuit breakers.

Date of creation: 2025-04-08

Author: Automatically generated by QuantalQ

Project Version: bac5c68

Link to Project: https://github.com/spring-petclinic/spring-petclinic-microservices

Contents

Toplevel-Documentation.md

Introduction

Overview

The Spring PetClinic Microservices application is a cloud-native version of the classic Spring PetClinic demo application. It demonstrates how to build a distributed system using Spring Cloud and microservices architecture principles. The application allows pet clinic staff to manage pet owners, their pets, veterinarians, and visit records through a web-based interface.

The system is decomposed into several microservices, each responsible for a specific business capability. These services communicate with each other through REST APIs and are designed to be independently deployable, scalable, and resilient. The architecture leverages modern cloud-native patterns such as service discovery, centralized configuration, API gateway, and circuit breakers.

Purpose of the Software

The Spring PetClinic Microservices application serves as a reference implementation for building cloud-native applications using Spring Boot and Spring Cloud. It demonstrates best practices for microservices architecture, including:

- Service decomposition based on business capabilities
- Inter-service communication patterns
- Centralized configuration management
- Service discovery and registration
- API gateway for request routing and aggregation
- Circuit breaking for resilience
- Monitoring and management of distributed services
- Integration of AI capabilities with traditional business services

The application enables pet clinic staff to efficiently manage information about pet owners, pets, veterinarians, and visit records, providing a comprehensive solution for pet clinic management.

Scope

The Spring PetClinic Microservices application includes the following capabilities:

- Management of pet owners and their contact information
- Management of pets and their details (name, birth date, type)
- Management of veterinarians and their specialties
- Recording and tracking of pet visits to the clinic
- AI-powered chat interface for answering questions about pet care
- Comprehensive monitoring and management of services

The application does not include:

- User authentication and authorization
- Billing or payment processing
- Appointment scheduling
- Inventory management
- Medical record details beyond basic visit information

Target Audience

This documentation is intended for:

- Software architects who need to understand the overall system design and architecture
- Software engineers who need to maintain or extend the application
- DevOps engineers responsible for deploying and operating the system
- Technical leaders evaluating microservices architecture patterns

- Product owners or managers who want to understand the technical capabilities of the system

Glossary

Term/Acronym	Full Form	Description
API	Application Programming Interface	A set of definitions and protocols for building and integrating application software
Circuit Breaker	-	A design pattern used to detect failures and prevent cascading failures in distributed systems
DTO	Data Transfer Object	An object that carries data between processes
JPA	Java Persistence API	A Java specification for accessing, persisting, and managing data between Java objects and a relational database
LLM	Large Language Model	AI model that processes and generates human-like text based on the input it receives
Microservice	-	An architectural style that structures an application as a collection of loosely coupled services
RAG	Retrieval-Augmented Generation	Technique that enhances LLM responses by retrieving relevant information from external sources
REST	Representational State Transfer	An architectural style for designing networked applications
Spring Boot	-	A framework that simplifies the development of Spring applications
Spring Cloud	-	Provides tools for developers to quickly build common patterns in distributed systems

Getting Started

Installation Guide

To install and run the Spring PetClinic Microservices application, follow these steps:

1. **Prerequisites**:
   - Java 17 or higher
   - Maven 3.6.0 or higher
   - Docker and Docker Compose (for containerized deployment)
   - Git (for cloning the repository)
2. **Clone the repository**:
   ```
   git clone https://github.com/spring-petclinic/spring-petclinic-microservices.git
   cd spring-petclinic-microservices
   ```
3. **Build the application**:
   ```
   ./mvnw clean install -DskipTests
   ```
4. **Run with Docker Compose**:

```
docker-compose up -d
```

5. **Alternative: Run services individually**:

```
./scripts/run_all.sh
```

6. **Access the application**: Open a web browser and navigate to http://localhost:8080

System Requirements

Minimum Requirements:

- **CPU**: 2 cores
- **Memory**: 4GB RAM
- **Disk**: 2GB free space
- **Operating System**: Any OS that supports Java 17 and Docker
- **Java**: JDK 17
- **Network**: Internet connection for building the application

Recommended Requirements:

- **CPU**: 4 cores or more
- **Memory**: 8GB RAM or more
- **Disk**: 5GB free space
- **Operating System**: Linux, macOS, or Windows 10/11
- **Java**: JDK 17
- **Network**: Broadband internet connection

Quick Start / First Run

After installation, the application will be accessible at http://localhost:8080. Here's how to get started:

1. **Explore the UI**:
   - The home page provides links to all main functions
   - Navigate to "FIND OWNERS" to manage pet owners
   - Navigate to "VETERINARIANS" to view the list of veterinarians
   - Use the "AI CHAT" feature to ask questions about pet care
2. **Add a new owner**:
   - Click on "FIND OWNERS"
   - Click "Add Owner"
   - Fill in the owner information and submit
3. **Add a pet to an owner**:
   - Find an owner in the system
   - Click "Add New Pet"
   - Fill in the pet information and submit
4. **Record a visit**:
   - Find an owner and select a pet
   - Click "Add Visit"
   - Enter the visit date and description
5. **Use the AI Chat**:
   - Click on "AI CHAT"
   - Type a question about pet care
   - Receive AI-generated responses based on veterinary knowledge

Prerequisites

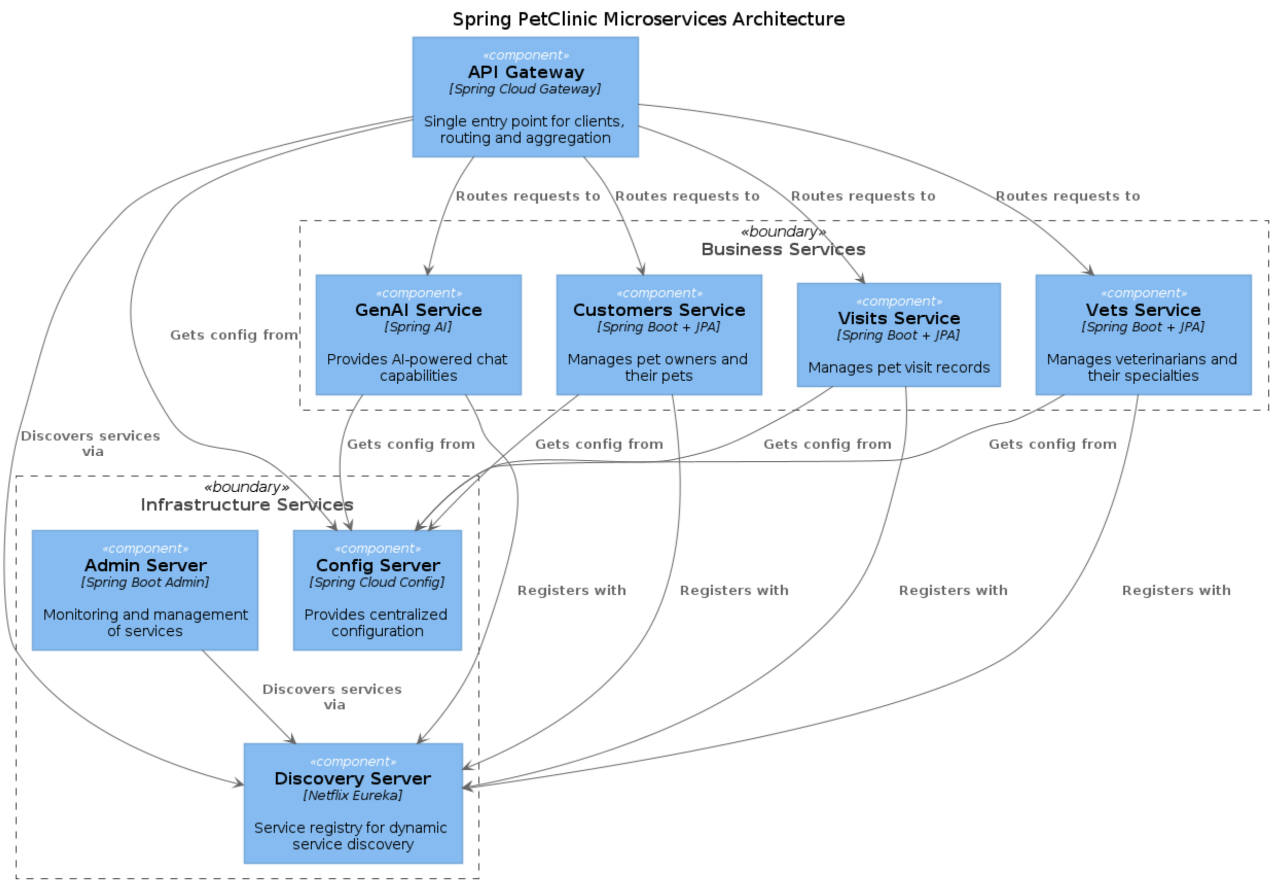
Before installing the Spring PetClinic Microservices application, ensure you have the following prerequisites:

- **Java Development Kit (JDK) 17**: Required for building and running the application
- **Maven**: Used for dependency management and building the project
- **Docker and Docker Compose**: Required for containerized deployment
- **Git**: Needed to clone the repository and for the Config Server
- **Internet Connection**: Required for downloading dependencies
- **Port Availability**: Ensure the following ports are available:
  - 8080: API Gateway
  - 8761: Discovery Server
  - 8888: Config Server
  - 9090: Admin Server
  - 8081-8085: Various microservices
  - 9411: Zipkin (distributed tracing)
  - 9090: Prometheus
  - 3000: Grafana

Architecture and Design Overview

System Architecture

The Spring PetClinic Microservices application follows a microservices architecture pattern, with services organized into infrastructure services and business services. The architecture is designed to demonstrate cloud-native patterns and practices.



The system architecture consists of the following key components:

1. **API Gateway**: Acts as a single entry point for all client requests, routing them to appropriate services and aggregating responses when necessary.
2. **Infrastructure Services**:

- **Config Server**: Provides centralized, externalized configuration for all microservices
- **Discovery Server**: Enables service registration and discovery using Netflix Eureka
- **Admin Server**: Provides monitoring and management capabilities for all services

3. **Business Services**:

- **Customers Service**: Manages pet owners and their pets
- **Vets Service**: Manages veterinarians and their specialties
- **Visits Service**: Manages pet visit records
- **GenAI Service**: Provides AI-powered chat capabilities using LLMs and RAG

For more detailed information about the architecture, please refer to the High-Level Documentation.

User Guide

Features Overview

The Spring PetClinic Microservices application provides the following key features:

1. **Owner Management**:
   - Create, update, and view pet owners
   - Search for owners by last name
   - View detailed owner information including their pets and visit history
2. **Pet Management**:
   - Add pets to owners
   - Update pet information (name, birth date, type)
   - View pet details and visit history
3. **Veterinarian Management**:
   - View list of veterinarians and their specialties
   - Filter veterinarians by specialty (UI feature)
4. **Visit Management**:
   - Schedule and record pet visits
   - View visit history for a specific pet
   - Track visit dates and descriptions
5. **AI-Assisted Support**:
   - Chat interface for asking questions about pet care
   - AI-generated responses based on veterinary knowledge
   - Integration with pet clinic data for context-aware responses

User Interface Guide

The application provides a web-based user interface accessible through the API Gateway. Here's an overview of the main screens:

1. **Home Page**:
   - Navigation menu for accessing all features
   - Welcome message and application overview
   - Links to key functions (Find Owners, Veterinarians, AI Chat)
2. **Find Owners Page**:
   - Search form for finding owners by last name
   - "Add Owner" button for creating new owners
   - List of matching owners when search is performed
3. **Owner Details Page**:
   - Owner information (name, address, city, telephone)
   - List of pets belonging to the owner
   - "Edit Owner" button for updating owner information
   - "Add New Pet" button for adding pets to the owner
   - Pet details including visit history
   - "Add Visit" button for scheduling visits for a pet

4. **New/Edit Owner Form**:
    - Form fields for owner information (first name, last name, address, city, telephone)
    - Validation for required fields
5. **New/Edit Pet Form**:
    - Form fields for pet information (name, birth date, type)
    - Dropdown for selecting pet type
    - Validation for required fields
6. **New Visit Form**:
    - Date picker for visit date
    - Text area for visit description
    - Validation for required fields
7. **Veterinarians Page**:
    - List of all veterinarians with their specialties
    - Sortable columns
8. **AI Chat Page**:
    - Chat interface with message history
    - Input field for typing questions
    - AI-generated responses displayed in conversation format

Step-by-Step Tutorials

Adding a New Owner

1. Click on "FIND OWNERS" in the navigation menu
2. Click the "Add Owner" button
3. Fill in the required information:
    - First Name
    - Last Name
    - Address
    - City
    - Telephone
4. Click "Add Owner"
5. You will be redirected to the owner details page for the newly created owner

Adding a Pet to an Owner

1. Find the owner by clicking "FIND OWNERS" and searching by last name
2. Click on the owner's name in the search results
3. On the owner details page, click "Add New Pet"
4. Fill in the pet information:
    - Name
    - Birth Date (use the date picker)
    - Type (select from dropdown)
5. Click "Add Pet"
6. The pet will appear in the list of pets on the owner details page

Recording a Visit

1. Find the owner and navigate to the owner details page
2. In the pets section, find the pet for which you want to record a visit
3. Click "Add Visit" for that pet
4. Enter the visit information:
    - Date (use the date picker)
    - Description of the visit
5. Click "Add Visit"
6. The visit will appear in the visit history for that pet

Using the AI Chat

1. Click on "AI CHAT" in the navigation menu
2. Type your question about pet care in the input field
3. Press Enter or click the send button
4. The AI will process your question and provide a response
5. Continue the conversation by asking follow-up questions
6. The chat maintains context of the conversation for more relevant responses

Usage Scenarios / Use Cases

Scenario 1: New Pet Registration **Actor**: Pet Clinic Staff

Flow: 1. Staff member searches for the owner by last name 2. If the owner doesn't exist, staff creates a new owner record 3. Staff adds a new pet to the owner with details (name, birth date, type) 4. Staff schedules an initial check-up visit for the pet 5. Staff verifies all information is correctly recorded

Outcome: New pet is registered in the system with owner information and initial visit scheduled.

Scenario 2: Veterinary Visit **Actor**: Pet Clinic Staff

Flow: 1. Owner brings pet for a scheduled visit 2. Staff searches for the owner by last name 3. Staff locates the pet in the owner's profile 4. Staff records the visit with date and description of procedures/diagnosis 5. Staff may consult the list of veterinarians for specialty information

Outcome: Visit is recorded in the system and associated with the pet.

Scenario 3: Pet Care Information Request **Actor**: Pet Owner or Pet Clinic Staff

Flow: 1. User navigates to the AI Chat interface 2. User asks a question about pet care (e.g., "How often should I vaccinate my dog?") 3. AI processes the question and retrieves relevant information 4. AI generates a response based on veterinary knowledge 5. User may ask follow-up questions for clarification

Outcome: User receives accurate information about pet care.

Troubleshooting Common Issues

Issue: Application doesn't start properly **Possible causes and solutions**: - **Service dependencies not starting in correct order**: Ensure you're using the provided scripts or Docker Compose to start services in the correct order - **Port conflicts**: Check if the required ports are already in use by other applications - **Insufficient memory**: Ensure your system meets the minimum requirements - **Java version mismatch**: Verify you're using Java 17 or higher

Issue: Cannot find an owner after adding them **Possible causes and solutions**: - **Search is case-sensitive**: Ensure you're entering the last name with the correct capitalization - **Service communication issue**: Check if the Customers Service is running and registered with the Discovery Server - **Database issue**: Verify the database is operational and accessible

Issue: AI Chat not responding or giving irrelevant answers **Possible causes and solutions**: - **GenAI Service not running**: Ensure the GenAI Service is started and registered with the Discovery Server - **Vector store not initialized**: The vector store might need time to initialize on first startup - **Question too ambiguous**: Try rephrasing your question to be more specific - **Network issues**: Check connectivity to external AI services if configured

Issue: Services showing as DOWN in Admin Server **Possible causes and solutions**: - **Service not started**: Ensure all services are started - **Network connectivity**: Check if services can communicate with each other - **Memory issues**: Verify services have sufficient memory allocated - **Health check failure**: Check the logs for specific health check failures

Developer Guide

Codebase Overview

The Spring PetClinic Microservices application is organized as a multi-module Maven project, with each module representing a separate microservice or shared component. The codebase follows Spring Boot and Spring Cloud conventions for structure and configuration.

Key aspects of the codebase:

1. **Module Structure**: Each microservice is a separate Maven module with its own dependencies, configuration, and packaging
2. **Spring Boot Applications**: Each service is a standalone Spring Boot application
3. **Spring Cloud Integration**: Services use Spring Cloud components for service discovery, configuration, and resilience
4. **REST APIs**: Services communicate through REST APIs defined using Spring Web
5. **Data Access**: JPA with Hibernate is used for data access in business services
6. **Testing**: JUnit and Spring Test are used for unit and integration testing

Folder Structure & Key Components

```
spring-petclinic-microservices/
 .github/ # GitHub configuration (Actions workflows)
 .mvn/ # Maven wrapper configuration
 docker/ # Docker configuration files
 docs/ # Documentation resources
 scripts/ # Utility scripts for running and deploying
 spring-petclinic-admin-server/ # Admin Server microservice
 spring-petclinic-api-gateway/ # API Gateway microservice
 spring-petclinic-config-server/ # Config Server microservice
 spring-petclinic-customers-service/ # Customers Service microservice
 spring-petclinic-discovery-server/ # Discovery Server microservice
 spring-petclinic-genai-service/ # GenAI Service microservice
 spring-petclinic-vets-service/ # Vets Service microservice
 spring-petclinic-visits-service/ # Visits Service microservice
 docker-compose.yml # Docker Compose configuration
 mvnw # Maven wrapper script (Unix)
 mvnw.cmd # Maven wrapper script (Windows)
 pom.xml # Parent POM file
 README.md # Project README
```

Each microservice module follows a similar structure:

```
spring-petclinic-xxx-service/
 src/
  main/
   java/ # Java source code
    org/springframework/samples/petclinic/xxx/
    application/ # Application services
    config/ # Configuration classes
    model/ # Domain model classes
    repository/ # Data repositories
    web/ # Web controllers
    XxxServiceApplication.java # Main class
   resources/
   application.yml # Application configuration
   db/ # Database scripts
   static/ # Static resources
  test/ # Test code
 pom.xml # Module POM file
```

Installation for Development

To set up a development environment:

1. **Clone the repository**:

   ```
   git clone https://github.com/spring-petclinic/spring-petclinic-microservices.git
   cd spring-petclinic-microservices
   ```

2. **Install dependencies**:

   ```
   ./mvnw clean install -DskipTests
   ```

3. **Import into IDE**:
   - For IntelliJ IDEA: File > Open > Select the root pom.xml > Open as Project
   - For Eclipse: File > Import > Maven > Existing Maven Projects > Select the root directory
4. **Run services individually**:
   - Run the Config Server first
   - Run the Discovery Server second
   - Run other services in any order
5. **Alternative: Use Docker for dependencies**:

   ```
   docker-compose up -d config-server discovery-server
   ```

   Then run only the services you're working on locally.

Build and Deployment Process

Building the Application To build the application:

```
./mvnw clean package
```

This will: 1. Compile the code 2. Run tests 3. Package each service as an executable JAR

To build Docker images:

```
./mvnw clean package -P buildDocker
```

This will: 1. Build the application 2. Create Docker images for each service

Deployment Options

1. **Local Deployment with Docker Compose**:

   ```
   docker-compose up -d
   ```

2. **Manual Deployment**:

   ```
   java -jar spring-petclinic-config-server/target/spring-petclinic-config-server.jar
   java -jar spring-petclinic-discovery-server/target/spring-petclinic-discovery-server.jar
   # Start other services similarly
   ```

3. **Kubernetes Deployment**: While not included in the repository, the Docker images can be deployed to Kubernetes using standard manifests or Helm charts.

Coding Standards and Conventions

The project follows standard Spring Boot and Java conventions:

1. **Java Code Style**:
   - Follow Oracle's Java code conventions
   - Use 4 spaces for indentation
   - Maximum line length of 120 characters
   - Use camelCase for variables and methods
   - Use PascalCase for classes and interfaces
2. **Package Structure**:
   - `org.springframework.samples.petclinic.<service>`: Root package for each service
   - Subpackages for different layers (model, repository, web, etc.)
3. **Naming Conventions**:
   - Controllers: Suffix with `Controller` or `Resource`
   - Services: Suffix with `Service`
   - Repositories: Suffix with `Repository`
   - Entities: No specific suffix
4. **REST API Conventions**:
   - Use plural nouns for resources (e.g., `/owners`, `/pets`)
   - Use HTTP methods appropriately (GET, POST, PUT, DELETE)
   - Return appropriate HTTP status codes
   - Use consistent response formats
5. **Documentation**:
   - Add Javadoc comments for public classes and methods
   - Use meaningful commit messages
   - Document REST APIs with appropriate annotations

API Documentation

The application exposes several REST APIs through its microservices. Here's a summary of the key APIs:

Customers Service API

Endpoint	Method	Description	Request Body	Response
`/owners`	GET	Get all owners	None	List of owners
`/owners/\{ownerId\}`	GET	Get owner by ID	None	Owner details
`/owners`	POST	Create new owner	Owner data	Created owner
`/owners/\{ownerId\}`	PUT	Update owner	Owner data	No content
`/owners/\{ownerId\}/pets`	POST	Add pet to owner	Pet data	Created pet
`/pets/\{petId\}`	GET	Get pet by ID	None	Pet details
`/pets/\{petId\}`	PUT	Update pet	Pet data	No content
`/petTypes`	GET	Get all pet types	None	List of pet types

Vets Service API

Endpoint	Method	Description	Request Body	Response
`/vets`	GET	Get all vets	None	List of vets with specialties

Visits Service API

Endpoint	Method	Description	Request Body	Response
`/owners/*/pets/\{petId\}/visits`	POST	Create new visit	Visit data	Created visit
`/owners/*/pets/\{petId\}/visits`	GET	Get visits for pet	None	List of visits
`/pets/visits`	GET	Get visits for multiple pets	Query params: petId (multiple)	Visits object with items

GenAI Service API

Endpoint	Method	Description	Request Body	Response
`/genai/chat`	POST	Send message to AI chat	Query text	AI response

API Gateway Routes The API Gateway routes requests to the appropriate microservices and provides aggregated endpoints:

Route	Target Service	Description
`/api/gateway/owners/\{ownerId\}`	Customers + Visits	Get owner with visits for their pets
`/api/customer/owners/**`	Customers	Forward to Customers Service
`/api/vet/vets/**`	Vets	Forward to Vets Service
`/api/visit/pets/**`	Visits	Forward to Visits Service
`/api/genai/**`	GenAI	Forward to GenAI Service

Database Schema and Interaction

The application uses a Database-per-Service pattern, where each service has its own database schema. Both HSQLDB (for development) and MySQL (for production) are supported.

Customers Service Schema

```
owners
- id (PK)
- first_name
- last_name
- address
- city
- telephone

pets
- id (PK)
- name
- birth_date
- type_id (FK to types)
- owner_id (FK to owners)

types
- id (PK)
- name
```

Vets Service Schema

```
vets
- id (PK)
- first_name
- last_name
```

```
specialties
- id (PK)
- name

vet_specialties
- vet_id (PK, FK to vets)
- specialty_id (PK, FK to specialties)
```

Visits Service Schema

```
visits
- id (PK)
- pet_id (FK to pets in Customers Service)
- visit_date
- description
```

Each service interacts with its database through Spring Data JPA repositories, which provide methods for CRUD operations and custom queries. The repositories use JPA entities mapped to the database tables.

Testing

Testing Strategy Overview

The Spring PetClinic Microservices application employs a comprehensive testing strategy that includes unit tests, integration tests, and end-to-end tests. The testing approach follows the testing pyramid principle, with more unit tests than integration tests, and fewer end-to-end tests.

The testing strategy aims to: - Ensure individual components work correctly in isolation - Verify that components integrate properly with each other - Validate that the system as a whole meets requirements - Provide regression protection for future changes

Types of Tests

Unit Tests Unit tests focus on testing individual components in isolation, with dependencies mocked or stubbed. Examples include:

- Testing controllers with MockMvc
- Testing repositories with an in-memory database
- Testing service classes with mocked dependencies

Key frameworks and tools used: - JUnit 5 for test execution - Mockito for mocking dependencies - AssertJ for fluent assertions - Spring Boot Test for testing Spring components

Example unit test for a controller:

```java
@WebMvcTest(VetResource.class)
class VetResourceTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private VetRepository vetRepository;

    @Test
    void shouldGetAVetInJSonFormat() throws Exception {
        // Given
        Vet vet = setupVet();
        given(vetRepository.findAll()).willReturn(Collections.singletonList(vet));

        // When/Then
        mockMvc.perform(get("/vets")
            .accept(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$[0].id").value(1))
            .andExpect(jsonPath("$[0].firstName").value("James"));
    }
}
```

Integration Tests Integration tests verify that different components work together correctly. Examples include:

- Testing REST controllers with a running Spring context
- Testing repositories with an actual database
- Testing service-to-service communication

Key frameworks and tools used: - Spring Boot Test for integration testing - TestContainers for database testing - WireMock for mocking external services

Example integration test:

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@ActiveProfiles("test")
class VisitsServiceIntegrationTest {

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    void shouldCreateVisit() {
        // Given
        Visit visit = new Visit();
        visit.setDate(new Date());
        visit.setDescription("Test␣Visit");
        visit.setPetId(1);

        // When
        ResponseEntity<Visit> response = restTemplate.postForEntity(
            "/owners/*/pets/1/visits", visit, Visit.class);

        // Then
        assertThat(response.getStatusCode()).isEqualTo(HttpStatus.CREATED);
        assertThat(response.getBody().getId()).isNotNull();
        assertThat(response.getBody().getPetId()).isEqualTo(1);
    }
}
```

End-to-End Tests End-to-end tests validate the entire system behavior from a user perspective. The application includes JMeter tests for performance and load testing.

Key frameworks and tools used: - JMeter for performance testing - Selenium (could be used for UI testing, not explicitly included)

Running Tests Locally

To run the tests locally:

1. **Run all tests**:

   ```
   ./mvnw test
   ```

2. **Run tests for a specific module**:

```
./mvnw test -pl spring-petclinic-customers-service
```

3. **Run a specific test class**:

```
./mvnw test -pl spring-petclinic-customers-service -Dtest=PetResourceTest
```

4. **Run with a specific profile**:

```
./mvnw test -P integration-tests
```

Continuous Integration & Testing

The application uses GitHub Actions for continuous integration and testing. The CI pipeline:

1. Builds the application
2. Runs unit tests
3. Runs integration tests
4. Reports test results

The CI configuration is defined in `.github/workflows/maven-build.yml`.

Known Issues and Test Results

The current test coverage varies across services:

- **Customers Service**: Good unit test coverage for controllers and repositories
- **Visits Service**: Basic controller tests implemented
- **Vets Service**: Controller tests implemented
- **API Gateway**: Limited test coverage
- **GenAI Service**: Limited test coverage

Areas for improvement: - Increase test coverage for the API Gateway - Add more integration tests for service-to-service communication - Implement end-to-end UI tests - Add more comprehensive performance tests

Configuration and Deployment

Configuration Management

The Spring PetClinic Microservices application uses Spring Cloud Config for centralized configuration management. This allows configuration to be externalized and managed separately from the application code.

Configuration Sources The Config Server serves configuration from:

1. **Git Repository**: The default configuration source (not explicitly configured in the example)
2. **Classpath**: Configuration files packaged with the Config Server
3. **File System**: Configuration files in a specified directory

Configuration Files Each service has its own configuration file named `<service-name>.yml`. Common configuration is defined in `application.yml`.

Example configuration structure:

```
application.yml # Common configuration for all services
customers-service.yml # Customers Service specific configuration
vets-service.yml # Vets Service specific configuration
visits-service.yml # Visits Service specific configuration
genai-service.yml # GenAI Service specific configuration
```

Configuration Properties Common configuration properties include:

- **Spring Cloud**: Service discovery, circuit breaker, and load balancing settings
- **Spring Data**: Database connection properties
- **Logging**: Log levels and patterns
- **Actuator**: Endpoints for monitoring and management
- **Service-Specific**: Properties specific to each service's functionality

Environment Variables

The application uses environment variables for configuration that varies between environments:

Variable	Description	Default Value
`SERVER\_PORT`	Port on which the service listens	Varies by service
`SPRING\_PROFILES\_ACTIVE`	Active Spring profiles	`default`
`SPRING\_CLOUD\_CONFIG\_URI`	URL of the Config Server	`http://localhost:8888`
`EUREKA\_CLIENT\_SERVICE\_URL\_DEFAULTZONE`	URL of the Eureka server	`http://localhost:8761/eureka/`
`SPRING\_DATASOURCE\_URL`	JDBC URL for database connection	Varies by service
`SPRING\_DATASOURCE\_USERNAME`	Database username	`petclinic`
`SPRING\_DATASOURCE\_PASSWORD`	Database password	`petclinic`

Deployment Guide

Docker Deployment The recommended way to deploy the application is using Docker Compose:

1. **Build the images** (if not using pre-built images):

```
./mvnw clean package -P buildDocker
```

2. **Deploy with Docker Compose**:

```
docker-compose up -d
```

3. **Verify deployment**:

```
docker-compose ps
```

4. **Access the application**: Open a web browser and navigate to http://localhost:8080

Manual Deployment For manual deployment:

1. **Start the Config Server**:

```
java -jar spring-petclinic-config-server/target/spring-petclinic-config-server.jar
```

2. **Start the Discovery Server**:

```
java -jar spring-petclinic-discovery-server/target/spring-petclinic-discovery-server.jar
```

3. **Start the Admin Server**:

```
java -jar spring-petclinic-admin-server/target/spring-petclinic-admin-server.jar
```

4. **Start the business services**:

```
java -jar spring-petclinic-customers-service/target/spring-petclinic-customers-service.jar
java -jar spring-petclinic-vets-service/target/spring-petclinic-vets-service.jar
java -jar spring-petclinic-visits-service/target/spring-petclinic-visits-service.jar
java -jar spring-petclinic-genai-service/target/spring-petclinic-genai-service.jar
```

5. **Start the API Gateway**:

```
java -jar spring-petclinic-api-gateway/target/spring-petclinic-api-gateway.jar
```

Scaling Considerations

The application is designed to be horizontally scalable:

1. **Stateless Services**: All services are designed to be stateless, allowing multiple instances to run concurrently
2. **Service Discovery**: Eureka enables dynamic discovery of service instances
3. **Load Balancing**: Client-side load balancing is provided by Spring Cloud LoadBalancer
4. **Database Scaling**: Each service has its own database, allowing databases to be scaled independently

To scale the application:

1. **Start additional instances** of a service on different ports or hosts
2. **Register with Eureka**: Each instance will automatically register with the Discovery Server
3. **Load Balancing**: Requests will be automatically distributed across available instances

Backup and Restore Procedures

For database backup and restore:

1. **HSQLDB** (development):
   - Backup: Copy the database files from the service's working directory
   - Restore: Replace the database files in the service's working directory
2. **MySQL** (production):
   - Backup:

     ```
     mysqldump -u petclinic -p petclinic > petclinic_backup.sql
     ```

   - Restore:

     ```
     mysql -u petclinic -p petclinic < petclinic_backup.sql
     ```

For configuration backup:

1. **Config Server** configuration:
   - Backup: Copy the configuration files or Git repository
   - Restore: Replace the configuration files or restore the Git repository

Integration and APIs

API Endpoints Documentation

The Spring PetClinic Microservices application exposes RESTful APIs through its various services. These APIs allow for integration with other systems and provide the foundation for the web UI.

Customers Service API

Endpoint	Method	Description	Request Body	Response
`/owners`	GET	Get all owners	None	List of owners
`/owners/\{ownerId \}`	GET	Get owner by ID	None	Owner details

Endpoint	Method	Description	Request Body	Response
`/owners`	POST	Create new owner	Owner data	Created owner
`/owners/\{ownerId \}`	PUT	Update owner	Owner data	No content
`/owners/\{ownerId \}/pets`	POST	Add pet to owner	Pet data	Created pet
`/pets/\{petId\}`	GET	Get pet by ID	None	Pet details
`/pets/\{petId\}`	PUT	Update pet	Pet data	No content
`/petTypes`	GET	Get all pet types	None	List of pet types

Example request/response for creating an owner:

Request:

```
POST /owners
{
  "firstName": "John",
  "lastName": "Doe",
  "address": "123 Main St",
  "city": "Boston",
  "telephone": "1234567890"
}
```

Response:

```
201 Created
{
  "id": 1,
  "firstName": "John",
  "lastName": "Doe",
  "address": "123 Main St",
  "city": "Boston",
  "telephone": "1234567890",
  "pets": []
}
```

Vets Service API

Endpoint	Method	Description	Request Body	Response
`/vets`	GET	Get all vets	None	List of vets with specialties

Example response:

```
200 OK
[
  {
    "id": 1,
    "firstName": "James",
    "lastName": "Carter",
    "specialties": []
  },
  {
    "id": 2,
    "firstName": "Helen",
    "lastName": "Leary",
    "specialties": [
      {
        "id": 1,
        "name": "radiology"
      }
    ]
  }
]
```

Visits Service API

Endpoint	Method	Description	Request Body	Response
`/owners/*/pets/\{petId\}/visits`	POST	Create new visit	Visit data	Created visit
`/owners/*/pets/\{petId\}/visits`	GET	Get visits for pet	None	List of visits
`/pets/visits`	GET	Get visits for multiple pets	Query params: petId (multiple)	Visits object with items

Example request/response for creating a visit:

Request:

```
POST /owners/*/pets/1/visits
{
  "date": "2023-06-01",
  "description": "Annual checkup"
}
```

Response:

```
201 Created
{
  "id": 1,
  "petId": 1,
  "date": "2023-06-01",
  "description": "Annual checkup"
}
```

GenAI Service API

Endpoint	Method	Description	Request Body	Response
`/genai/chat`	POST	Send message to AI chat	Query text	AI response

Example request/response:

Request:

```
POST /genai/chat
{
  "query": "How often should I vaccinate my dog?"
}
```

Response:

```
200 OK
{
  "response": "Dogs typically need core vaccines every 1-3 years, depending on the vaccine type and
      your local regulations. Core vaccines usually include rabies, distemper, parvovirus, and
      adenovirus. Puppies need a series of vaccinations starting at 6-8 weeks of age, with boosters
      every 3-4 weeks until 16 weeks old. After the initial puppy series, a booster is given at one
      year of age, then every 1-3 years depending on the vaccine and your vet's recommendation. Always
       consult with your veterinarian for a vaccination schedule tailored to your dog's specific needs
       and lifestyle."
}
```

External Integrations and Dependencies

The Spring PetClinic Microservices application has the following external dependencies:

1. **Spring Cloud Services**:
   - Spring Cloud Config for configuration management
   - Netflix Eureka for service discovery
   - Spring Cloud Gateway for API gateway functionality
   - Spring Boot Admin for monitoring and management
2. **Databases**:
   - HSQLDB for development
   - MySQL for production
3. **Monitoring Tools**:
   - Prometheus for metrics collection
   - Grafana for metrics visualization
   - Zipkin for distributed tracing
4. **AI Services**:
   - The GenAI Service may integrate with external AI providers (configuration details not specified)

Authentication and Authorization

Information not available: The current implementation does not include authentication or authorization mechanisms. In a production environment, security measures such as OAuth2/OpenID Connect should be implemented.

Webhooks and Callback Interfaces

Information not available: The application does not implement webhooks or callback interfaces in the current version.

Data Exchange Formats

The application uses JSON as the primary data exchange format for all APIs. All requests and responses are formatted as JSON, with the following conventions:

1. **Date Format**: ISO 8601 format (YYYY-MM-DD)
2. **Naming Convention**: camelCase for property names
3. **Content Type**: `application/json`
4. **Character Encoding**: UTF-8

Example of a pet representation:

```
{
  "id": 1,
  "name": "Leo",
  "birthDate": "2020-09-07",
  "type": {
    "id": 1,
    "name": "cat"
  },
  "owner": {
    "id": 1,
    "firstName": "George",
    "lastName": "Franklin"
  },
  "visits": [
    {
      "id": 1,
      "date": "2021-03-04",
      "description": "rabies shot"
    }
  ]
}
```

Security

Security Assets

The following table summarizes the key security assets identified across the application:

Service	Asset	Description	Security Measure	Assessment of Criticality
Config Server	Configuration Data	Application configuration including potential sensitive information	Served over HTTPS when properly configured	Critical - should ensure HTTPS is used in production
Config Server	Git Repository	Source of configuration data	Uses HTTPS for secure communication	Uncritical - public repository
Config Server	Server Port	Port 8888 used to access the Config Server	No explicit security measures in the examined code	Critical - should be protected by a firewall in production
GenAI Service	LLM API Credentials	Credentials for accessing the LLM API	Assumed to be stored in environment variables or configuration	Information not available: No explicit handling visible in the code
GenAI Service	Customer Data	Personal information of pet owners	Transmitted over secure connections between services	Uncritical - No storage of sensitive data in this service
GenAI Service	Conversation History	Chat history stored in memory	Stored in memory only, not persisted	Uncritical - Limited to 10 messages per conversation
API Gateway	Service Communication	Communication between API Gateway and backend services	Protected by circuit breaker for fault tolerance	Moderate - Provides resilience but not security
All Services	Service Ports	Ports used by various microservices	No explicit security measures mentioned	Critical - Should be protected in production

Security Guidelines

Information not available: The current implementation does not include explicit security guidelines. In a production environment, the following security measures should be implemented:

1. **Authentication and Authorization**:

   - Implement OAuth2/OpenID Connect for user authentication
   - Use Spring Security for authorization
   - Secure service-to-service communication
2. **Data Protection**:
   - Use HTTPS for all communications
   - Encrypt sensitive data at rest
   - Implement proper input validation
3. **Network Security**:
   - Use firewalls to restrict access to service ports
   - Implement rate limiting to prevent DoS attacks
   - Configure proper network segmentation
4. **Monitoring and Logging**:
   - Implement security event logging
   - Monitor for suspicious activities
   - Set up alerts for security incidents

Data Privacy Considerations

Information not available: The current implementation does not include explicit data privacy considerations. In a production environment, the following measures should be implemented:

1. **Personal Data Handling**:
   - Identify and classify personal data
   - Implement data minimization principles
   - Provide mechanisms for data subject rights
2. **Data Retention**:
   - Define and implement data retention policies
   - Implement secure data deletion procedures
   - Document data flows and storage locations
3. **Consent Management**:
   - Implement mechanisms for obtaining and managing consent
   - Provide clear privacy notices
   - Allow users to withdraw consent

Vulnerability Management

Information not available: The current implementation does not include explicit vulnerability management procedures. In a production environment, the following measures should be implemented:

1. **Dependency Management**:
   - Regularly update dependencies to address known vulnerabilities
   - Use tools like OWASP Dependency Check to scan for vulnerabilities
   - Implement a process for emergency patching
2. **Security Testing**:
   - Implement security testing in the CI/CD pipeline
   - Conduct regular security assessments
   - Perform penetration testing
3. **Incident Response**:
   - Develop and maintain an incident response plan
   - Establish roles and responsibilities for security incidents
   - Conduct regular drills and reviews

Authentication and Authorization Mechanisms

Information not available: The current implementation does not include authentication or authorization mechanisms. In a production environment, the following mechanisms should be implemented:

1. **User Authentication**:

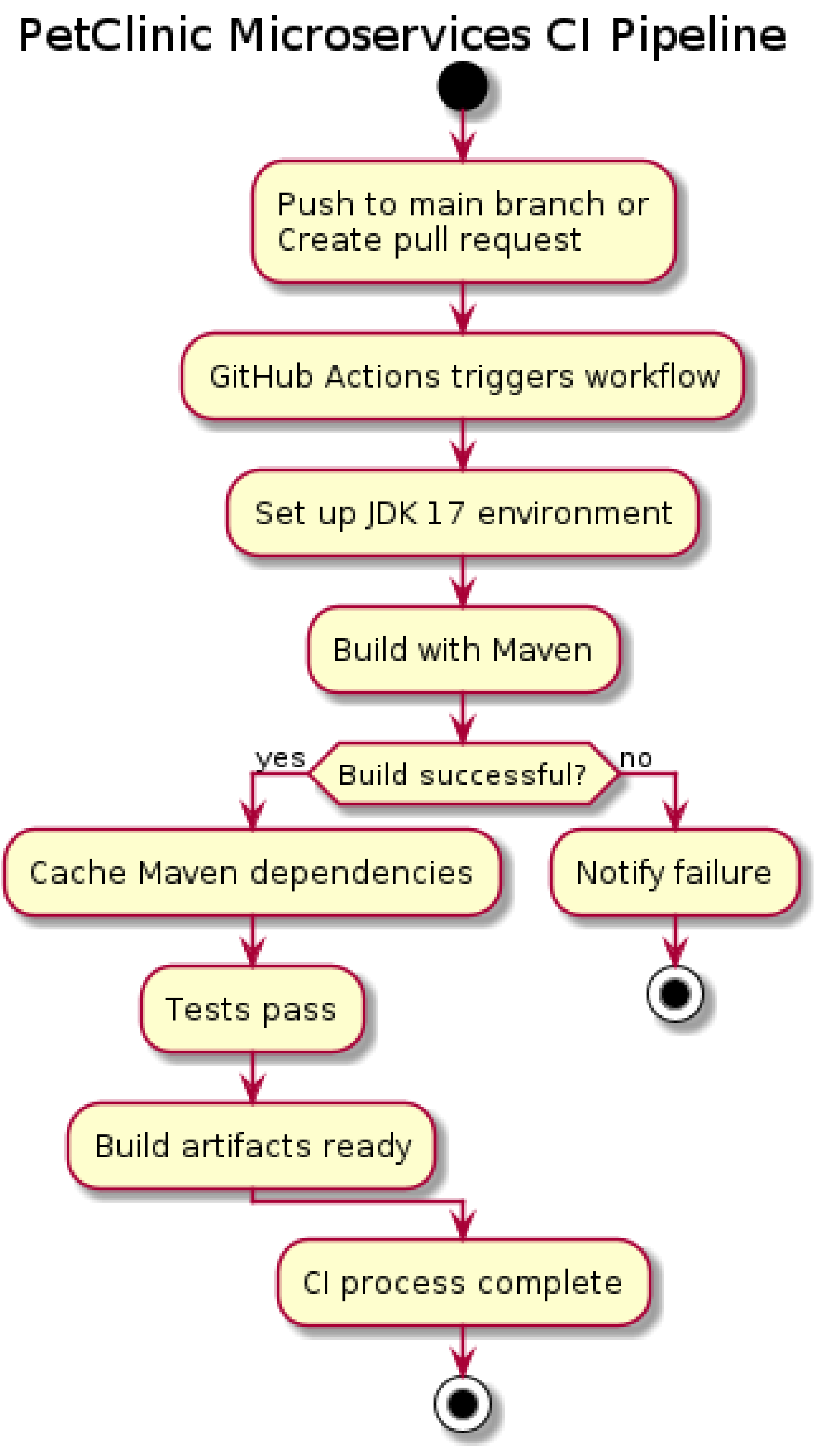
    - Implement OAuth2/OpenID Connect for user authentication
    - Use secure password storage with bcrypt or similar algorithms
    - Implement multi-factor authentication for sensitive operations
2. **Authorization**:
    - Implement role-based access control
    - Apply the principle of least privilege
    - Use Spring Security for authorization enforcement
3. **API Security**:
    - Secure APIs with API keys or OAuth2 tokens
    - Implement proper CORS configuration
    - Use CSRF protection for web forms

DevOps and CI/CD Pipeline

The Spring PetClinic Microservices application implements a modern DevOps approach with continuous integration, containerization, and deployment automation. This section describes the DevOps practices and tools used in the project.

Continuous Integration

The application uses GitHub Actions for continuous integration, with the workflow defined in `.github/workflows/maven-build.yml`:


PetClinic Microservices CI Pipeline
Push to main branch or Create pull request
GitHub Actions triggers workflow
Set up JDK 17 environment
Build with Maven
Build successful?
yes
no
Cache Maven dependencies
Notify failure
Tests pass
Build artifacts ready
CI process complete

CI Workflow Details:

- **Trigger**: The CI workflow is triggered on push to the main branch and pull requests targeting the main branch
- **Environment**: Runs on Ubuntu with Java 17
- **Build Process**: Uses Maven to build the application
- **Caching**: Implements caching of Maven dependencies to speed up subsequent builds

- **Commands**: Executes `mvn -B package --file pom.xml` to build and package the application

Build System

The application uses Maven as its build system, with a multi-module project structure:

- **Parent POM**: Defines common dependencies, plugins, and properties for all modules
- **Maven Profiles**:
  - `springboot`: Activated for Spring Boot applications, configures the Spring Boot Maven plugin
  - `buildDocker`: Used to build Docker images for each service

Key Maven Plugins:

1. **Spring Boot Maven Plugin**: Builds executable JARs and provides build information
2. **Git Commit ID Maven Plugin**: Includes Git information in the build
3. **Maven Enforcer Plugin**: Ensures the correct Java version is used
4. **Exec Maven Plugin**: Used for Docker image building

Containerization

The application provides comprehensive Docker support for containerized deployment:

Docker Configuration:

1. **Dockerfile**: Located in the `docker` directory, uses a multi-stage build:
   - First stage: Extracts application layers from the Spring Boot JAR
   - Second stage: Creates a lightweight runtime image with Eclipse Temurin JDK 17
2. **Docker Compose**: The `docker-compose.yml` file defines all services:
   - Infrastructure services (Config Server, Discovery Server, Admin Server)
   - Business services (Customers, Vets, Visits, GenAI)
   - Monitoring services (Prometheus, Grafana, Zipkin)
   - Resource limits for each service
3. **Docker Build Process**:
   - Configured in the `buildDocker` Maven profile
   - Supports both Docker and Podman as container runtimes
   - Builds multi-architecture images (linux/amd64, linux/arm64)
   - Configurable image tags and repositories

Deployment Automation

The project includes several scripts in the `scripts` directory to automate deployment tasks:

1. `run_all.sh`: Starts all services locally with proper sequencing:
   - Starts infrastructure services (Prometheus, Grafana, Zipkin)
   - Launches application services in the correct order
   - Configures chaos monkey for resilience testing
2. `pushImages.sh`: Pushes Docker images to a container registry:
   - Uses environment variables for repository prefix and version
   - Pushes all service images
3. `tagImages.sh`: Tags Docker images with version information:
   - Prepares images for release
   - Uses environment variables for versioning
4. **Chaos Testing**: The `scripts/chaos` directory contains scripts for chaos engineering:
   - Enables/disables chaos monkey attacks
   - Configures different failure scenarios (exceptions, latency, memory issues)
   - Monitors different components for resilience testing

Monitoring and Observability

The DevOps pipeline integrates monitoring and observability tools:

1. **Prometheus**: Collects metrics from all services
2. **Grafana**: Visualizes metrics with pre-configured dashboards
3. **Spring Boot Admin**: Provides a web UI for monitoring and managing services
4. **Zipkin**: Implements distributed tracing

DevOps Pipeline Flow

The complete DevOps pipeline follows this flow:

1. **Development**: Developers work on features in branches
2. **Continuous Integration**: GitHub Actions builds and tests code on push/PR
3. **Build**: Maven builds the application and creates artifacts
4. **Containerization**: Docker images are built for each service
5. **Deployment**: Services are deployed using Docker Compose or scripts
6. **Monitoring**: Prometheus, Grafana, and Spring Boot Admin monitor the application

For production environments, additional steps would be required, such as deploying to Kubernetes or other orchestration platforms, but these are not explicitly defined in the current codebase.

Change Log and Release Notes

Versioning Scheme

Information not available: The documentation does not provide explicit information about the versioning scheme used by the Spring PetClinic Microservices application. Based on common practices, it likely follows Semantic Versioning (SemVer):

- **Major version**: Incremented for incompatible API changes
- **Minor version**: Incremented for backward-compatible functionality additions
- **Patch version**: Incremented for backward-compatible bug fixes

Release History

Information not available: The documentation does not provide a complete release history for the Spring PetClinic Microservices application.

Notable Changes

Information not available: The documentation does not provide information about notable changes in different releases of the Spring PetClinic Microservices application.

License and Legal Information

Software Licensing

The Spring PetClinic Microservices application is open source software licensed under the Apache License 2.0. This is a permissive license that allows users to:

- Use the software for any purpose
- Distribute the software
- Modify the software
- Distribute modified versions of the software
- Use the software commercially

The full license text is available in the LICENSE file in the repository.

Contribution Guidelines

Information not available: The documentation does not provide explicit contribution guidelines for the Spring PetClinic Microservices application.

Copyright Notices

Appendix

References

- Spring PetClinic Microservices GitHub Repository
- Spring Boot Documentation
- Spring Cloud Documentation
- Spring Data JPA Documentation
- Spring AI Documentation
- Netflix Eureka Wiki
- Resilience4j Documentation

Additional Resources and Further Reading

- Microservices with Spring Boot and Spring Cloud
- Building Microservices: Designing Fine-Grained Systems
- Spring Cloud Service Discovery with Eureka
- Centralized Configuration with Spring Cloud Config
- Spring Boot Actuator: Production-ready Features
- Retrieval-Augmented Generation (RAG)
- Large Language Models (LLMs)