Modern Software Development Practices

Agile Development with Containers and Cloud-Native Technologies

Red Hat Open Innovation Labs

Table of contents

1. Modern Software Development Practices

Developed by Red Hat Open Innovation Labs for IndiGo Airlines

1.1 Mission

Empowering teams to build cutting-edge software through agile development practices and cloud-native technologies.

This comprehensive guide brings together the essential practices, tools, and methodologies needed to deliver high-quality software in today's fast-paced development landscape. We focus on leveraging **containerization**, **agile workflows**, and **open-source technologies** to enable rapid iteration, reliable deployments, and scalable architectures.

1.2 What You'll Learn

- **Agile development practices** that accelerate delivery and improve collaboration
- **Containerization with Podman** for consistent, reproducible development and deployment environments
- **Cloud-native architectures** that leverage microservices, 12-factor principles, and modern design patterns
- **Test-driven development** and continuous integration/deployment workflows
- **Enterprise-grade solutions** using Red Hat UBI (Universal Base Images) and ecosystem tools

Our approach emphasizes **practical application** with real-world examples using **.NET and React Native**, all containerized with **Podman and podman compose**.

1.3 Documentation Structure

1.3.1 1. Cloud Fundamentals

Master cloud-native concepts, hybrid cloud strategies, and the CNCF landscape—the foundation for building modern, scalable applications.

1.3.2 2. Containerization

Learn container fundamentals with Podman, CRI-O container engines, and orchestration with Kubernetes/OpenShift to create portable, consistent environments.

1.3.3 3. Development Foundations

Adopt proven practices: requirement-centric development, 12-factor app methodology, microservices architecture, clean architecture, and optimized inner-loop workflows.

1.3.4 4. Agile Development & Testing

Implement BDD/TDD practices, local development with podman compose, API mocking strategies, and performance testing in containerized environments.

1.3.5 5. Red Hat Ecosystem

Leverage Red Hat Software Collections and UBI images for secure, enterprise-grade container development with supply chain integrity.

1.4 Getting Started

New to cloud-native development? Start with **Cloud Fundamentals** to understand the paradigm shift.

Ready to containerize? Jump to **Containerization** to master Podman and container workflows.

Building applications? Explore **Development Foundations** for architectural patterns and agile methodologies.

Focused on quality? Dive into **Agile Development & Testing** for BDD/TDD and testing strategies.

Deploying to production? Leverage the **Red Hat Ecosystem** for secure, enterprise-grade container images.

1.5 Why This Approach?

Speed & Agility: Containerization and agile practices reduce friction between development, testing, and production, enabling faster iteration cycles.

Consistency: Podman and UBI images ensure your development environment matches production, eliminating "works on my machine" issues.

Quality & Reliability: Test-driven development, clean architecture, and continuous integration practices catch issues early and maintain code quality.

Security & Compliance: Red Hat UBI images provide enterprise-grade security, supply chain transparency, and long-term support.

Open Source: All tools and technologies are built on open-source foundations, giving you flexibility and community support.

1.6 How to Use This Guide

Each section is structured for hands-on learning: - **Clear explanations** of concepts and their practical value - **Working examples** using .NET and React Native with Podman - **Step-by-step instructions** you can follow immediately - **Common pitfalls** and how to avoid them - **Best practices** for production-ready code - **Hands-on exercises** to reinforce learning

Use the sidebar navigation to explore topics in sequence or jump directly to what your team needs right now.

1.7 Technologies Covered

- **Containers**: Podman, podman compose, CRI-O
- **Orchestration**: Kubernetes, OpenShift
- **Languages/Frameworks**: .NET (C#), React Native
- **Base Images**: Red Hat UBI 8/9
- **Testing**: BDD, TDD, performance testing
- **Architecture**: Microservices, 12-factor apps, clean architecture

1.8 Copyright & Usage

Created by: Red Hat Open Innovation Labs
For: IndiGo Airlines

This documentation is proprietary IndiGo Airlines. It may not be copied, distributed, modified, or shared outside of IndiGo Airlines without express written permission from Indigo Airlines.

This documentation is continuously updated to reflect the latest practices in agile development and cloud-native software engineering. All examples are tested with the specified tools and versions.

2. 1. Cloud Fundamentals

2.1 Hybrid Cloud

Leveraging vendor specific cloud features may lead to:

- Increased difficulty in migrating workloads to other cloud providers
- Higher long-term costs due to lack of competition or pricing flexibility
- Limited portability and interoperability of applications
- Greater dependency on a single vendor's roadmap and support
- Potential compliance or data residency challenges if features are not available in all regions

By being aware of these risks, organizations can make more informed decisions about when and how to use proprietary cloud features, and consider hybrid or multi-cloud strategies to maintain flexibility and control. * **Dependency on provider roadmap**: Changes or discontinuation of services by the provider can disrupt business operations. * **Compliance and data residency issues**: Some features may not be available in all regions, impacting regulatory compliance. * **Potential for higher long-term costs**: Initial incentives may give way to higher ongoing expenses as workloads grow.

2.1.1 Hybrid Cloud Approach

In a hybrid cloud strategy we aim to combine on-premises infrastructure, private cloud, and public cloud services to create a unified, consitent flexible computing environment. This approach allows organizations to:

- Avoid vendor lock-in by designing applications for portability and interoperability.
- Meet regulatory or data residency requirements by keeping sensitive workloads on-premises or in private clouds.
- Optimize costs by running workloads in the most appropriate environment.
- Enhance resilience and business continuity by distributing workloads across multiple platforms.
- Reuse the code across multiple vendor solutions.

Advantages of a Hybrid Cloud Approach:

- Enables the use of open standards and cloud-agnostic tools (such as Kubernetes and Terraform) for consistent infrastructure management.
- Increases application portability by minimizing reliance on proprietary services, making it easier to move workloads between environments.
- Enhances security and connectivity through robust networking and security controls across on-premises, private, and public clouds.
- Provides unified monitoring and management of resources, helping optimize cost, performance, and compliance across all environments.

Real world example

EXAMPLE: UNIFIED CI/CD WITH TEKTON ACROSS HYBRID CLOUD PROVIDERS

In a hybrid cloud environment, organizations often use different CI/CD solutions depending on where their code is hosted or deployed. For example, you might use:

- **GitLab CI** for projects hosted on GitLab (on-premises or cloud)
- **GitHub Actions** for repositories on GitHub
- **Tekton/OpenShift Pipelines** for cloud-native, Kubernetes-based deployments

Challenge:
Each of these CI/CD systems has its own configuration syntax and pipeline definitions. This means that if you move your code or workloads between providers (e.g., from GitHub to GitLab, or from on-premises to the cloud), you often have to rewrite your CI/CD pipeline code to fit the new system.

Solution with Tekton:

Tekton is a cloud-native, Kubernetes-based CI/CD solution that uses standard Kubernetes resources to define pipelines. By adopting Tekton, you can write your pipeline definitions once and run them anywhere Kubernetes is available—on-premises, in the public cloud, or in a hybrid environment—without changing your pipeline code.

2.2 A Developer's Guide to Cloud Native

2.2.1 Overview

To unlock the full potential of the cloud, developers must build applications designed specifically for that environment. **Cloud native development (CND)** is the approach for doing just that. It involves creating applications as a collection of independent services, packaged in containers, and managed by automated systems. For any developer working today, understanding these core principles is essential for building scalable, resilient, and modern software that realizes true business value.

CND involves architecting applications as collections of loosely coupled services, using containers for packaging, and leveraging cloud platforms for orchestration, scaling, and management.

The Four Pillars of Cloud Native

1. **Microservices Architecture**: Applications built as independent, loosely coupled services and well alinged with business outcomes.
2. **Containerization**: Applications packaged in lightweight, portable containers.
3. **DevOps and CI/CD**: Automated build, test, and deployment pipelines for rapid, reliable releases.
4. **Dynamic Orchestration**: Automated management of containers using tools like Kubernetes and service meshes.

Cloud Native Characteristics

- **Scalability**: Applications can scale horizontally across distributed infrastructure.
- **Resilience**: Built-in fault tolerance and recovery mechanisms.
- **Observability**: Comprehensive monitoring, logging, and tracing capabilities.
- **Automation**: Infrastructure and deployment are automated and managed as code.
- **Portability**: Applications run consistently across different cloud environments.

Cloud Native vs. Traditional Development

Aspect	Traditional	Cloud Native
Architecture	Monolithic	Microservices
Deployment	Manual/Scripted	Automated CI/CD
Scaling	Vertical	Horizontal
Infrastructure	Static/Physical	Dynamic/Virtual
State Management	Stateful	Stateless preferred
Communication	Direct calls	API-first

2.2.2 Key Cloud Native Concepts

Horizontal Scaling

Horizontal scaling (also called "scaling out") is the process of adding more instances of an application or service to handle increased load, rather than increasing the resources (CPU, RAM) of a single instance (vertical scaling). In cloud native environments, this is typically achieved by deploying

additional containers or microservice replicas, often managed automatically by orchestration platforms like Kubernetes. Horizontal scaling enables applications to efficiently handle variable workloads and provides high availability.

Example:

> If a web application experiences a spike in traffic, Kubernetes can automatically start more container instances (pods) to distribute the load, then scale them down when demand decreases.

Monitoring & Observability

Monitoring in cloud native systems involves continuously collecting, analyzing, and visualizing metrics, logs, and traces from applications and infrastructure. This enables teams to detect issues, understand system health, and optimize performance. True **observability** is the goal, allowing you to ask arbitrary questions about your system's state without having to pre-define all monitoring dashboards.

Common tools: - **Prometheus** for metrics collection and alerting - **Grafana** for visualization - **OpenTelemetry** for distributed tracing

Best practices: - Monitor both application and infrastructure metrics. - Set up automated alerts for anomalies. - Use dashboards for real-time visibility.

Distributed Transaction Management

In a microservices architecture, a single business process may span multiple services and databases, making traditional (ACID) transactions difficult. **Distributed transaction management** refers to techniques for ensuring data consistency and reliability across these services.

Common patterns: - **Saga Pattern:** Breaks a transaction into a series of local transactions, coordinated through events or commands. If a step fails, compensating actions are triggered to undo previous steps. - **Two-Phase Commit (2PC):** A classic protocol for distributed transactions, but less common in cloud native due to its complexity and performance trade-offs.

Key considerations: - Prefer eventual consistency over strict consistency where possible. - Design idempotent operations and compensating transactions. - Use message queues or event buses for coordination.

Centralized Logging

Centralized logging aggregates logs from all application components and infrastructure into a single system. This makes it dramatically easier to search, analyze, and troubleshoot issues in distributed cloud native environments.

Benefits: - Simplifies debugging across multiple services. - Enables correlation of events and tracing of requests. - Supports compliance and auditing requirements.

Typical stack: - **Fluentd** or **Logstash** for log collection and forwarding - **Elasticsearch** for log storage and indexing - **Kibana** or **Grafana Loki** for log visualization and querying

Best practices: - Structure logs (e.g., JSON format) for easier parsing. - Include contextual information (request IDs, user IDs, service names). - Set up log retention and access controls.

3. 2. Containerization

3.1 Containers: Citizens of the Hybrid Cloud

3.1.1 Background

Traditionally, software releases involved promoting application code and its configuration across separate environments. This often led to inconsistencies between development, testing, and production, resulting in failed deployments and the classic "it worked on my machine" problem.

Containers solve this issue by bundling the application with all its libraries and dependencies into a single, portable unit. This guarantees that the application runs identically everywhere, from a developer's laptop to the production servers. By eliminating environmental drift, containers enable enterprises to release software more reliably, innovate faster, and scale with confidence.

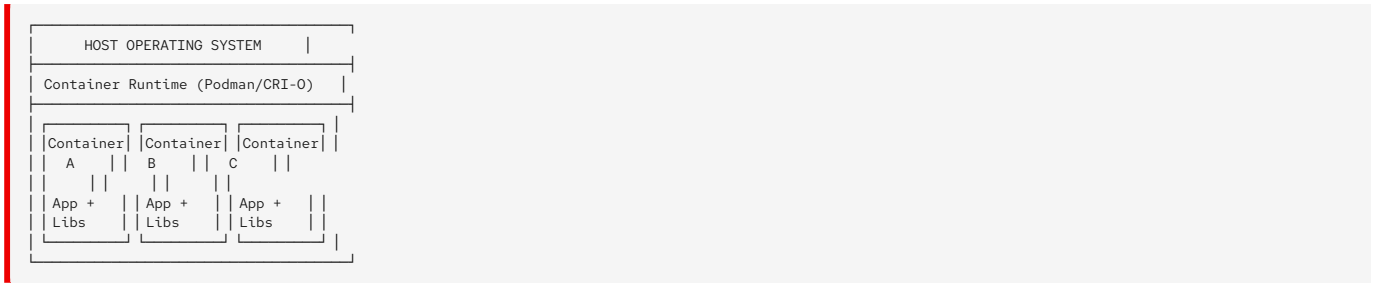
3.1.2 What is a Container?

Think of a container as a process like any other software, but with its own underlying operating system environment. Running an OS as a program on a host machine opens up several possibilities. For example, if a tool only works on Alma Linux or RHEL, we can use an appropriate container without impacting other processes running on the system.

For application developers, they can now start visualization that their software too can run on the desired OS and the runtime tools they want without impacting the host machine or other processes.

A typical container encapsulates everything needed to run the application—code, runtime, system tools, libraries, and configuration files—into a standardized package. As a result, containers guarantee reliable operation across different environments, from a developer's laptop to production servers.

Visualization



Container Process Isolation

A container operates as an isolated process on the host system with the following characteristics:

1. **Namespace Isolation**: Provides dedicated views of system resources including process IDs, network interfaces, and file systems
2. **File System Virtualization**: Each container maintains an independent file system view through layered images
3. **Resource Management**: Control groups (cgroups) enforce resource limits for CPU, memory, and I/O operations
4. **Security Boundaries**: Multiple isolation mechanisms prevent interference between containers and the host system

This process-based approach enables containers to share the host OS kernel while maintaining application isolation and security.

3.1.3 Container Images

A running application is typically called a process or service. In the world of containers, the definition for what a container will run comes from a container image. A container image acts as a blueprint: it contains all the code, dependencies, and configuration needed to create a running container. When executed, this image provides everything required for the containerized application to function consistently across environments.

Images and Containers Relationship

Container technology distinguishes between two fundamental concepts:

- **Container Image**: A static, immutable template containing application code, runtime components, system tools, libraries, and configuration settings
- **Container**: A running instance of a container image executing as a process on the host system

This relationship follows a template-instance pattern where:

- **Image**: Serves as the immutable template defining the container's contents and configuration
- **Container**: Represents the runtime instantiation of the image as an active process

Layered File System Architecture

Container images implement a layered file system architecture that provides operational and management benefits:

```
┌──────────────────────────────────────┐
│     Application Layer            │ ← Application code
├──────────────────────────────────────┤
│    Dependencies Layer            │ ← Package dependencies
├──────────────────────────────────────┤
│    Runtime Layer                 │ ← Runtime environment
├──────────────────────────────────────┤
│    Base OS Layer                 │ ← Red Hat UBI
└──────────────────────────────────────┘
          Container Image
```

LAYER ARCHITECTURE DIAGRAM

graph TD A[Base Layer - Red Hat UBI] --> B[Runtime Layer - .NET 6.0] B --> C[Dependencies Layer - NuGet Packages] C --> D[Application Layer - Your Code] E[Base Layer - Red Hat UBI] --> F[Runtime Layer - Node.js 18] F --> G[Dependencies Layer - npm Packages] G --> H[Application Layer - React Native App] A -.->|Shared| E style A fill:#e1f5fe style E fill:#e1f5fe style B fill:#f3e5f5 style F fill:#f3e5f5 style C fill:#e8f5e8 style G fill:#e8f5e8 style D fill:#fff3e0 style H fill:#fff3e0

Image Formats and Standards

DOCKER FORMAT

The **Docker container format** is the original and most widely recognized standard for packaging and distributing containerized applications. Developed by Docker, Inc., this format popularized the use of containers in modern software development. For many, the terms "container" and "Docker" have become nearly synonymous, reflecting Docker's dominant role in the early container ecosystem.

However, it's important to note that the Docker image format and runtime were initially closed source and tightly coupled to Docker's own tooling and platform. This proprietary approach led to concerns about vendor lock-in and interoperability, motivating the industry to develop open standards like the OCI (Open Container Initiative) format. RedHat encourges everyone to stick to open source standards.

OCI (OPEN CONTAINER INITIATIVE) STANDARD

The container industry has adopted the OCI Image Format as the standard specification, ensuring compatibility across different container runtimes and platforms:

```
{
  "schemaVersion": 2,
  "mediaType": "application/vnd.oci.image.manifest.v1+json",
  "config": {
    "mediaType": "application/vnd.oci.image.config.v1+json",
    "size": 1469,
    "digest": "sha256:83f19d5902f8b2d8..."
  },
  "layers": [
    {
      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
      "size": 73109,
      "digest": "sha256:b49b96bfa4aa2..."
    }
  ]
}
```

BASE IMAGES

Container base images are the foundational layers upon which all containerized applications are built. Choosing a stable and secure base image from a trusted source is critical because vulnerabilities or misconfigurations in the base can compromise the entire application stack. Trusted sources, such as official or vendor-supported registries, provide regularly updated images with security patches and compliance guarantees, reducing the risk of introducing security flaws and ensuring consistent, reliable deployments across environments.

RED HAT UNIVERSAL BASE IMAGES (UBI)

Red Hat Universal Base Images provide enterprise-grade container base images for production deployments:

```
UBI Image Family:
├── ubi8/ubi              (Full base image)
├── ubi8/ubi-minimal      (Minimal base - 92MB)
├── ubi8/ubi-micro        (Ultra minimal - 37MB)
├── ubi8/dotnet-60        (.NET 6.0 runtime)
├── ubi8/nodejs-18        (Node.js 18 runtime)
└── ubi8/python-39        (Python 3.9 runtime)
```

UBI Advantages:

- **Enterprise Security**: Regular security updates and vulnerability patches
- **Compliance**: Adherence to enterprise security and compliance standards
- **Support**: Integration with Red Hat enterprise support services
- **Redistribution Rights**: Freely redistributable without licensing restrictions
- **CVE Management**: Proactive security vulnerability identification and remediation

3.1.4 Container Repositories and Distribution

Container Registries

Container registries serve as centralized repositories for storing and distributing container images. These registries support the software supply chain by providing:

- **Image Storage**: Centralized storage for container images with version control
- **Access Control**: Authentication and authorization mechanisms for image access
- **Distribution**: Efficient image distribution across different environments
- **Security Scanning**: Automated vulnerability assessment of stored images

Red Hat Container Catalog

The Red Hat Container Catalog provides enterprise-grade container images with:

- **Certified Images**: Verified compatibility with Red Hat platforms
- **Security Updates**: Regular security patches and vulnerability fixes
- **Support Integration**: Enterprise support for containerized applications
- **Compliance Standards**: Adherence to enterprise security and compliance requirements

Image Naming and Tagging

Container images follow standardized naming conventions for identification and versioning:

```
registry.access.redhat.com/ubi8/dotnet-60:latest
├──────────────────────────┤ ├──┤ ├────────┤ ├────┤
│            ||  ||    ||  |
│   Registry Host      ||  Namespace ||Tag|
│            ||  || Image ||  |
└──────────────────────────┘ └──┘ └────────┘ └────┘
```

Naming Components: - **Registry**: Container registry hostname (registry.access.redhat.com) - **Namespace**: Organizational or project grouping (ubi8) - **Repository**: Specific image name (dotnet-60) - **Tag**: Version or variant identifier (latest, v1.0, dev)

3.1.5 Container Image Examples

Example 1: .NET Application Container Image

This example demonstrates the layered structure of a .NET application container image:

```
# Base layer: Red Hat UBI .NET runtime
FROM registry.access.redhat.com/ubi8/dotnet-60-runtime

# Application configuration layer
WORKDIR /app
EXPOSE 5000
ENV ASPNETCORE_URLS=http://+:5000
ENV ASPNETCORE_ENVIRONMENT=Production

# Security layer: Non-root user creation
RUN groupadd -r appuser && useradd -r -g appuser appuser

# Application layer: Copy application files
COPY --chown=appuser:appuser bin/Release/net6.0/publish/ .
USER appuser

# Runtime configuration
ENTRYPOINT ["dotnet", "MyApi.dll"]
```

Layer Breakdown: - **Base Layer**: UBI .NET runtime environment (shared across .NET applications) - **Configuration Layer**: Environment variables and port configuration - **Security Layer**: User account and permission setup - **Application Layer**: Application-specific code and dependencies

Example 2: Multi-Stage Build Pattern

Multi-stage builds optimize image size by separating build and runtime environments:

```
# Build stage
FROM registry.access.redhat.com/ubi8/dotnet-60 AS build
WORKDIR /src
COPY *.csproj .
RUN dotnet restore
COPY . .
RUN dotnet publish -c Release -o /app/publish

# Runtime stage
FROM registry.access.redhat.com/ubi8/dotnet-60-runtime AS runtime
WORKDIR /app
COPY --from=build /app/publish .
RUN groupadd -r appuser && useradd -r -g appuser appuser
RUN chown -R appuser:appuser /app
USER appuser
ENTRYPOINT ["dotnet", "MyApi.dll"]
```

Benefits: - **Reduced Size**: Final image contains only runtime dependencies - **Security**: Build tools excluded from production image - **Performance**: Faster container startup and network transfer

3.1.6 Container Design Considerations

Image Size Optimization

Container image size impacts performance, security, and resource utilization:

Optimization Strategies:

1. **Minimal Base Images**: Use purpose-built minimal images (ubi-minimal, ubi-micro)
2. **Multi-Stage Builds**: Separate build and runtime environments
3. **Layer Consolidation**: Combine related operations in single RUN instructions
4. **Dependency Management**: Include only required runtime dependencies

Image Size Comparison:

```
Image Type               Size        Use Case
ubi8/ubi                214MB       Full development environment
ubi8/ubi-minimal         92MB       Production applications
ubi8/ubi-micro           37MB       Microservices and minimal applications
```

Security Considerations

Container security requires attention to multiple layers:

Security Principles:

1. **Least Privilege**: Run applications with minimal required permissions
2. **Non-Root Users**: Avoid running processes as root within containers
3. **Minimal Attack Surface**: Include only necessary components and dependencies
4. **Regular Updates**: Maintain current base images with security patches

Security Implementation Example:

```
FROM registry.access.redhat.com/ubi8/ubi-minimal

# Create dedicated application user
RUN microdnf install shadow-utils && \
    groupadd -r appuser && \
    useradd -r -g appuser appuser && \
    microdnf clean all

WORKDIR /app
COPY --chown=appuser:appuser application /app/
USER appuser

ENTRYPOINT ["./application"]
```

3.1.7 Best Practices

Security Best Practices

1. **Use Rootless Containers**: Run Podman without root privileges
2. **Non-root User**: Always run applications as non-root users inside containers
3. **Minimal Base Images**: Use UBI minimal images when possible
4. **Regular Updates**: Keep base images updated with security patches
5. **Secrets Management**: Never include secrets in images

```
# Example of secure container setup
FROM registry.access.redhat.com/ubi8/dotnet-60-runtime

# Create non-root user
RUN groupadd -r appuser && useradd -r -g appuser appuser

WORKDIR /app
COPY --chown=appuser:appuser . .

# Switch to non-root user
USER appuser

ENTRYPOINT ["dotnet", "MyApp.dll"]
```

Performance Best Practices

1. **Layer Caching**: Order containerfile instructions to maximize cache hits
2. **Multi-stage Builds**: Separate build and runtime environments
3. **Minimize Layers**: Combine RUN commands when logical
4. **.containerignore**: Exclude unnecessary files from build context

Example `.containerignore`:

```
node_modules
.git
*.md
.gitignore
containerfile*
*.log
```

Development Workflow

1. **Local Development**: Use volume mounts for hot reloading
2. **Testing**: Create separate images for testing environments
3. **Debugging**: Use interactive shells for troubleshooting

```
# Interactive debugging session
podman run -it --rm myimage /bin/bash

# Run with development volume mounts
podman run -d \
  -v $(pwd):/app:Z \
  -p 5000:5000 \
  --name dev-container \
  myapi:dev
```

3.1.8 Tools and Resources

Essential Podman Commands

```
# Image management
podman images                    # List images
podman pull image:tag           # Pull image
podman rmi image:tag            # Remove image
podman build -t name:tag .      # Build image

# Container management
podman ps                       # List running containers
podman ps -a                    # List all containers
podman run [options] image      # Run container
podman stop container           # Stop container
podman rm container             # Remove container

# Networking
podman network ls               # List networks
podman network create name      # Create network
podman port container           # Show port mappings

# Troubleshooting
podman logs container           # View logs
podman exec -it container bash  # Interactive shell
podman inspect container        # Detailed information
```

Useful Resources

- Podman Official Documentation
- Red Hat UBI Images
- Container Best Practices
- Rootless Containers Guide

3.1.9 Hands-on Exercise

Exercise: Containerize a Full-Stack Application

Create a multi-container application with: 1. .NET Web API backend 2. React frontend 3. PostgreSQL database

Requirements: - Use only Red Hat UBI base images - Implement proper security practices (non-root users) - Use Podman networks for container communication - Include proper error handling and logging - Create a podman compose file for orchestration

Steps:

1. Create the .NET API with Entity Framework
2. Create a React frontend that consumes the API
3. Set up PostgreSQL with proper initialization
4. Create containerfiles for each component
5. Create a `compose.yml` file
6. Test the complete application stack

Deliverables: - Working containerized application - Documentation of the containerization process - Security analysis of the implementation - Performance testing results

3.1.10 Summary

Container fundamentals with Podman provide the foundation for modern application development and deployment. Key takeaways include:

- **Podman offers enhanced security** through rootless containers and daemonless architecture
- **Red Hat UBI images** provide enterprise-grade security and compliance
- **Proper security practices** include non-root users and minimal base images
- **Multi-stage builds** optimize image size and security
- **Container orchestration** enables complex application architectures
- **Development workflows** benefit from containerization through consistency and portability

By mastering these container fundamentals, developers can build secure, scalable, and maintainable applications that follow modern DevOps practices and integrate seamlessly with cloud-native architectures.

3.2 Container Engines: Podman and CRI-O

3.2.1 Overview

Container engines provide the runtime environment necessary for creating, managing, and executing containers.

While Docker Dekstop and Docker container engine is more popular there are few open source alternatices. Podman and CRI-O are two leading open source container engines, especially in the Red Hat ecosystem and Kubernetes environments.

Podman

Podman is a daemonless, open source container engine developed by Red Hat. It is compatible with the OCI (Open Container Initiative) standards and provides a Docker-compatible command-line interface. Key features include:

- **Daemonless architecture**: No central daemon; each command runs in its own process, improving security and reliability.
- **Rootless containers**: Users can run containers without root privileges, reducing the attack surface.
- **Docker CLI compatibility**: Most Docker commands work with Podman, making migration easy.
- **Pod management**: Supports Kubernetes-style pods natively.
- **Integration with systemd**: Easily generate systemd unit files for managing containers as services.

Podman is ideal for development, CI/CD pipelines, and production environments where security and compatibility are priorities.

CRI-O

CRI-O is a lightweight container runtime specifically designed for Kubernetes. It implements the Kubernetes Container Runtime Interface (CRI) and uses OCI-compliant container images. Key features include:

- **Minimal footprint**: Only provides the features required by Kubernetes, reducing complexity and attack surface.
- **Kubernetes native**: Integrates tightly with Kubernetes, making it a preferred runtime for OpenShift and other Kubernetes distributions.
- **Security**: Supports SELinux, seccomp, AppArmor, and other Linux security features.
- **Stability and performance**: Focuses on reliability and efficient resource usage.

CRI-O is not intended for direct use by developers; instead, it is used by Kubernetes nodes to manage containers.

Buildah

While Podman and CRI-O focus on running and managing containers, **Buildah** is a specialized tool for building OCI and Docker container images. Buildah is also developed by Red Hat and is designed to work seamlessly with Podman.

BUILDAH

- **Purpose-built for image building**: Buildah provides a flexible command-line interface for creating, building, and modifying container images.
- **Daemonless and rootless**: Like Podman, Buildah does not require a daemon and supports rootless operation.
- **Scriptable and composable**: Buildah commands can be used in shell scripts and CI/CD pipelines for fine-grained control over image creation.
- **OCI and Docker compatibility**: Produces images that are compatible with both OCI and Docker standards.
- **Integration with Podman**: Podman can use Buildah under the hood for building images (`podman build`).

Buildah is ideal for advanced image-building workflows, automation, and scenarios where you need more control than a traditional Containerfile provides.

SKOPEO

Another related tool is **Skopeo**, which is used for copying, inspecting, and signing container images between different registries and storage backends, without requiring a local container runtime.

- **Copy images**: Move images between registries, local storage, and more.
- **Inspect images**: View image metadata without pulling the image.
- **Image signing and verification**: Supports container image security workflows.

When to Use Each

- **Podman**: For running, managing, and orchestrating containers and pods, both locally and in production.
- **CRI-O**: As the container runtime for Kubernetes clusters.
- **Buildah**: For building and customizing container images, especially in automated or rootless environments.
- **Skopeo**: For image management tasks such as copying, inspecting, and signing images.

3.3 Overview

Previously, we explored containers and their many benefits, such as portability, consistency, and resource efficiency. Most developers are comfortable running containers locally on their laptops for development and testing. However, deploying containers at enterprise scale—where you may need to run hundreds or thousands of containers to support business applications—introduces new challenges.

Running containers on a single machine or manually managing them across multiple virtual machines quickly becomes inefficient, error-prone, and difficult to scale. Enterprises need robust solutions to orchestrate, manage, and automate containerized workloads across clusters of servers.

3.3.1 How to Run Containers at Scale?

To run containers at scale, organizations use **container orchestration platforms**. These platforms automate the deployment, scaling, networking, and management of containers across clusters of machines. The most widely adopted orchestration platform is **Kubernetes**.

What is Kubernetes?

Kubernetes (often abbreviated as K8s) is an open-source system for automating deployment, scaling, and management of containerized applications. It provides:

- **Automated scheduling**: Efficiently places containers based on resource requirements and constraints.
- **Self-healing**: Restarts failed containers, replaces and reschedules them when nodes die, and kills containers that don't respond to health checks.
- **Horizontal scaling**: Automatically increases or decreases the number of container instances based on demand.
- **Service discovery and load balancing**: Exposes containers using DNS names or IP addresses and distributes network traffic.
- **Rolling updates and rollbacks**: Updates applications with zero downtime and can revert to previous versions if needed.
- **Secret and configuration management**: Manages sensitive information and application configuration separately from code.

Kubernetes abstracts away the underlying infrastructure, allowing you to run containers consistently across on-premises, public cloud, or hybrid environments.

3.3.2 OpenShift and Its Benefits

While Kubernetes is powerful, it can be complex to set up and manage, especially for enterprise needs such as security, compliance, and developer productivity. This is where **OpenShift** comes in.

What is OpenShift?

OpenShift is an enterprise Kubernetes platform developed by Red Hat. It builds on top of Kubernetes, adding features and tools to simplify and secure container orchestration for organizations. OpenShift is available as both an open-source project (**OKD**) and a commercial offering (**Red Hat OpenShift Container Platform**).

Key Benefits of OpenShift

- **Enterprise-Grade Security**: Built-in security policies, role-based access control (RBAC), and integrated authentication/authorization.
- **Developer Productivity**: Source-to-Image (S2I) builds, integrated CI/CD pipelines, and developer-friendly web consoles.
- **Integrated Monitoring and Logging**: Out-of-the-box tools for monitoring, logging, and alerting.
- **Multi-Tenancy**: Isolates workloads and resources for different teams or projects.
- **Automated Operations**: Simplifies cluster installation, upgrades, and management with automation tools.
- **Hybrid and Multi-Cloud Support**: Deploy and manage applications across on-premises, public cloud, or hybrid environments.
- **Operator Framework**: Automates the management of complex, stateful applications on Kubernetes.

When to Use OpenShift?

OpenShift is ideal for organizations that want the power and flexibility of Kubernetes, but with enhanced security, developer tools, and enterprise support. It is widely used in regulated industries, large enterprises, and organizations seeking to accelerate their cloud-native journey.

Running containers in many forms

Kubernetes and OpenShift provide several different workload types, allowing you to run containers in ways that best fit your application's requirements. These workload types are defined as Kubernetes resources, each designed for specific use cases:

1. DEPLOYMENT

A **Deployment** is the most common way to run stateless applications. It manages a set of identical pods, ensures the desired number of replicas are running, and supports rolling updates and rollbacks. Deployments are ideal for web servers, APIs, and other scalable, stateless services.

2. STATEFULSET

A **StatefulSet** is used for running stateful applications that require stable network identities, persistent storage, and ordered deployment or scaling. Examples include databases (like PostgreSQL, MongoDB) and distributed systems (like Kafka, Zookeeper). StatefulSets ensure each pod has a unique, stable identity and persistent volume.

3. JOB

A **Job** creates one or more pods to run a task to completion. Once the task finishes successfully, the job is marked as complete. Jobs are perfect for batch processing, data migrations, or any workload that needs to run to completion rather than continuously.

4. CRONJOB

A **CronJob** runs jobs on a scheduled basis, similar to cron in Linux. This is useful for periodic tasks such as backups, report generation, or scheduled data processing.

5. DAEMONSET

A **DaemonSet** ensures that a copy of a pod runs on every (or selected) node in the cluster. This is commonly used for cluster-wide services like log collection, monitoring agents, or networking components.

6. REPLICASET

A **ReplicaSet** ensures a specified number of pod replicas are running at any given time. While Deployments use ReplicaSets under the hood, you can use ReplicaSets directly for lower-level control, though this is less common.

OpenShift fully supports all these Kubernetes workload types and adds its own abstractions (like BuildConfig and DeploymentConfig) for enhanced developer workflows. This flexibility allows you to choose the right pattern for your application—whether it's a stateless web service, a persistent database, a scheduled batch job, or a system-level agent.

Summary Table:

Workload Type	Use Case	Example
Deployment	Stateless, scalable apps	Web servers, APIs
StatefulSet	Stateful apps needing stable identity/storage	Databases, Kafka
Job	Run-to-completion tasks	Data migration, batch jobs
CronJob	Scheduled jobs	Nightly backups, reports
DaemonSet	One pod per node	Log collectors, monitoring agents

By leveraging these workload types, Kubernetes and OpenShift enable you to run containers in the way that best matches your application's needs, ensuring scalability, reliability, and operational efficiency.

3.3.3 Next Steps

- Learn the basic concepts and architecture of Kubernetes (pods, deployments, services, etc.).
- Explore how OpenShift builds on Kubernetes and provides additional value.
- Try deploying a simple application on a local OpenShift or Kubernetes cluster (e.g., using Minikube or CodeReady Containers).
- Dive deeper into advanced topics like CI/CD integration, security, and multi-cloud deployments with OpenShift.

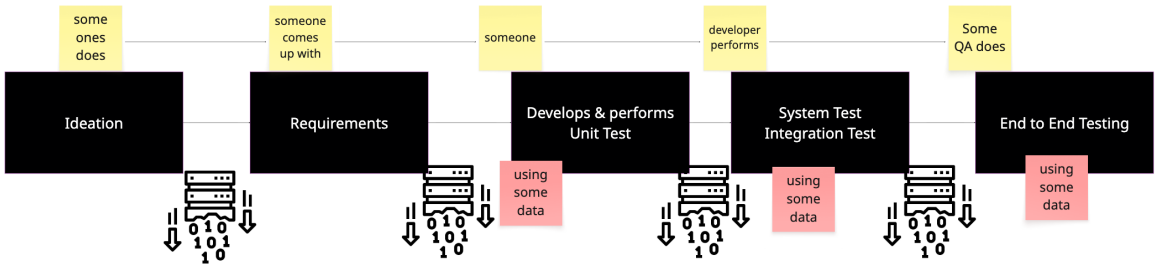
For hands-on labs and further reading, see the resources below:

- Kubernetes Official Documentation
- OpenShift Documentation
- OKD (OpenShift Origin) Community

4. 3. Development Foundations

4.1 Overview

A typical software development cycle often follows a linear, stage-gated process:



This traditional approach highlights two major challenges:

- **Requirements Drift:** There is a significant gap between the initial ideation/requirements phase and the final end-to-end testing. As the project progresses, the original requirements can become diluted or misinterpreted.
- **Information Loss:** With each handoff between stages (requirements, design, development, testing), some information or intent is lost or altered, increasing the risk that the delivered product does not fully align with the original vision.

How can we overcome these issues? This is where the **Requirement-Centric Development** approach comes into play.

4.1.1 What is Requirement-Centric Development?

Requirement-Centric Development is a modern approach that places requirements at the heart of the entire software development lifecycle. Instead of treating requirements as a one-time, up-front activity, this approach ensures that requirements are continuously referenced, validated, and refined throughout all stages of development and testing.

In practice, teams implement this with BDD and TDD: BDD expresses requirements as executable Gherkin scenarios, and TDD drives code to satisfy them (see BDD and TDD).

Key Principles

- **Traceability:** Every feature, user story, test case, and piece of code is directly traceable back to a specific requirement.
- **Continuous Validation:** Requirements are validated early and often, not just at the end. This reduces the risk of late-stage surprises.
- **Collaboration:** Stakeholders, developers, testers, and product owners collaborate around requirements, ensuring shared understanding and alignment.
- **Living Documentation:** Requirements are treated as living artifacts that evolve with feedback and learning, rather than static documents.

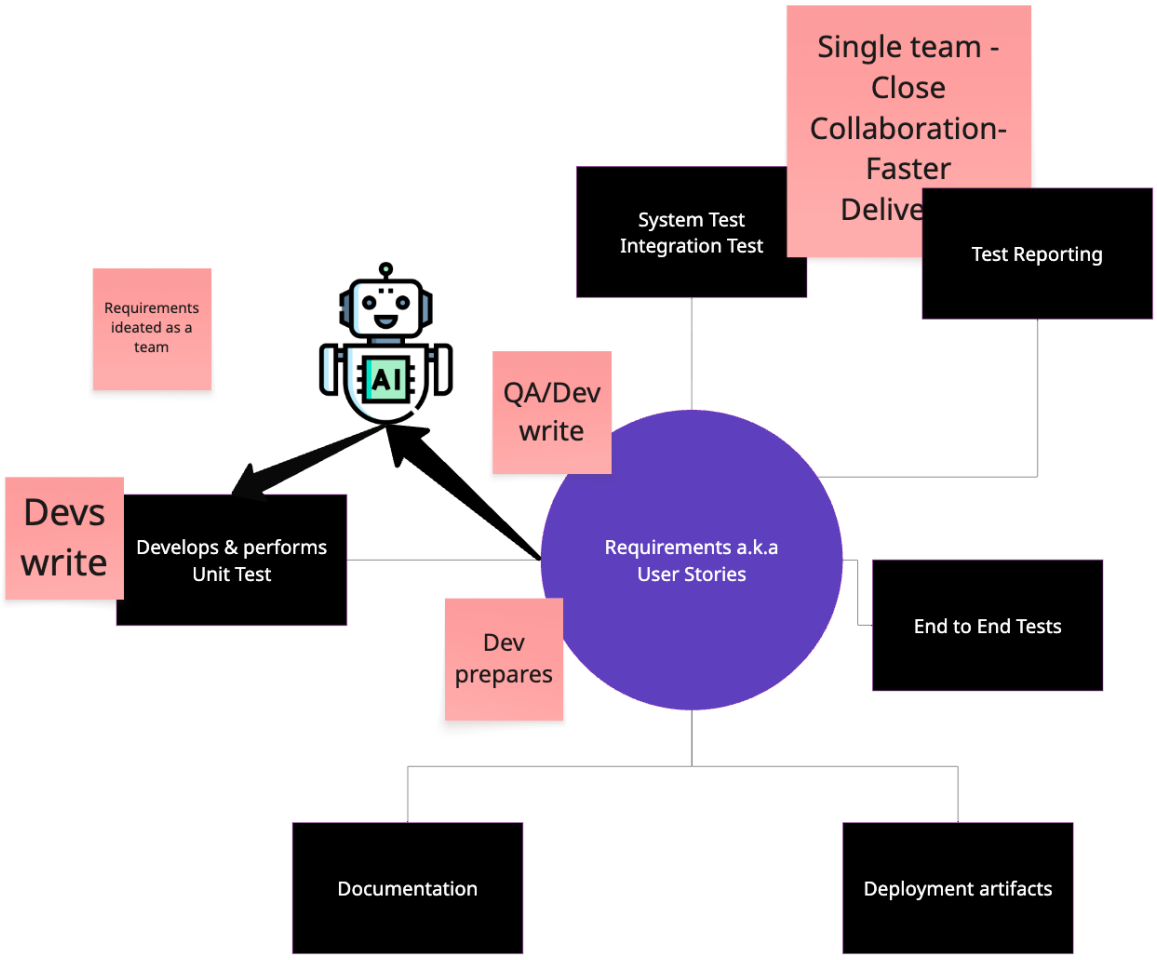
How It Works

The requirement-centric approach integrates requirements into every phase:

1. **Requirements Elicitation:** Gather and document clear, testable requirements with stakeholder input.
2. **Design & Development:** Design and implement features with direct reference to requirements, ensuring alignment.
3. **Automated Testing:** Derive test cases directly from requirements, enabling automated validation and regression testing.
4. **Continuous Feedback:** Use test results and stakeholder feedback to refine requirements and implementation iteratively.

4.1.2 Requirement-Centric Development Lifecycle

The following diagram illustrates how requirements remain central throughout the process:



4.1.3 Benefits

- **Reduced Risk of Misalignment:** By keeping requirements central, the risk of delivering the wrong product is minimized.
- **Faster Feedback Loops:** Early and continuous validation catches issues sooner.
- **Improved Quality:** Test cases derived from requirements ensure comprehensive coverage.

- **Greater Agility:** Requirements can evolve as new insights are gained, supporting adaptive planning.

4.2 Overview

Modern application development—especially for cloud-native, containerized, and hybrid cloud environments—benefits significantly from following the 12-Factor App methodology. Originally designed for building software-as-a-service (SaaS) applications, the 12-Factor approach has become a foundational set of best practices for creating scalable, maintainable, and portable applications that thrive in dynamic infrastructure environments.

4.2.1 What Are the 12 Factors?

The 12 factors are a set of guidelines that address key aspects of application design and deployment, including:

1. **Codebase** – One codebase tracked in revision control, many deploys
2. **Dependencies** – Explicitly declare and isolate dependencies
3. **Config** – Store configuration in the environment
4. **Backing Services** – Treat backing services as attached resources
5. **Build, Release, Run** – Strictly separate build and run stages
6. **Processes** – Execute the app as one or more stateless processes
7. **Port Binding** – Export services via port binding
8. **Concurrency** – Scale out via the process model
9. **Disposability** – Maximize robustness with fast startup and graceful shutdown
10. **Dev/Prod Parity** – Keep development, staging, and production as similar as possible
11. **Logs** – Treat logs as event streams
12. **Admin Processes** – Run admin/management tasks as one-off processes

By adhering to these principles, developers can ensure their applications are resilient, easy to configure, and ready for continuous integration and deployment (CI/CD). The 12-Factor methodology encourages statelessness, environment-agnostic configuration, and seamless integration with cloud-native platforms such as Kubernetes and OpenShift.

4.2.2 Why 12-Factor Matters for Cloud-Native and Containers

- **Portability:** 12-Factor apps can be deployed on any cloud or container platform with minimal changes.
- **Scalability:** Statelessness and process-based concurrency make it easy to scale horizontally.
- **Maintainability:** Clear separation of concerns and configuration management simplifies updates and troubleshooting.
- **DevOps Alignment:** The methodology aligns with modern DevOps practices, enabling rapid iteration and reliable deployments.

4.2.3 Further Reading and References

- The Twelve-Factor App (Official Site)
- Red Hat: Building 12-Factor Applications
- Red Hat Developer: 12-Factor App Principles for Cloud-Native Development
- Red Hat OpenShift: 12-Factor App Best Practices

4.3 Microservices Architecture

4.3.1 Overview

Before we develop containerized applications, we must also understand the key architecture behind this.

Microservices architecture is a modern approach to building applications as a suite of small, independently deployable services, each running in its own process and communicating via lightweight mechanisms such as HTTP APIs or messaging. This design enables teams to develop, deploy, and scale components independently, fostering agility and resilience.

4.3.2 Microservices Architecture Diagram

graph TB subgraph "Client Layer" WEB[Web App] MOBILE[Mobile App] API_GW[API Gateway] end subgraph "Service Layer" USER_SVC[User Service] ORDER_SVC[Order Service] PAYMENT_SVC[Payment Service] INVENTORY_SVC[Inventory Service] NOTIFICATION_SVC[Notification Service] end subgraph "Data Layer" USER_DB[(User DB)] ORDER_DB[(Order DB)] PAYMENT_DB[(Payment DB)] INVENTORY_DB[(Inventory DB)] end subgraph "Infrastructure" MSG_BROKER[Message Broker
Apache Kafka/RabbitMQ] SERVICE_MESH[Service Mesh
Istio/Linkerd] CONFIG[Config Server] REGISTRY[Service Registry] end WEB --> API_GW MOBILE --> API_GW API_GW --> USER_SVC API_GW --> ORDER_SVC API_GW --> PAYMENT_SVC API_GW --> INVENTORY_SVC USER_SVC --> USER_DB ORDER_SVC --> ORDER_DB PAYMENT_SVC --> PAYMENT_DB INVENTORY_SVC --> INVENTORY_DB ORDER_SVC -.->|async events| MSG_BROKER PAYMENT_SVC -.->|async events| MSG_BROKER INVENTORY_SVC -.->|async events| MSG_BROKER NOTIFICATION_SVC -.->|async events| MSG_BROKER SERVICE_MESH -.->|service discovery| REGISTRY USER_SVC -.->|config| CONFIG ORDER_SVC -.->|config| CONFIG PAYMENT_SVC -.->|config| CONFIG INVENTORY_SVC -.->|config| CONFIG

Bounded Contexts

A core concept in microservices is the **bounded context**—each service encapsulates a specific business capability and its own data model. This clear separation reduces coupling, minimizes dependencies, and allows teams to evolve services independently. Bounded contexts are often aligned with business domains, ensuring that each microservice has a well-defined responsibility and interface.

4.3.3 Communication Patterns

Synchronous Communication

HTTP/REST APIs - Direct service-to-service communication - Request-response pattern - Suitable for real-time queries and commands - Can introduce tight coupling and latency issues

sequenceDiagram participant Client participant API Gateway participant Order Service participant Payment Service Client->>API Gateway: POST /orders API Gateway->>Order Service: Create Order Order Service->>Payment Service: Process Payment Payment Service-->>Order Service: Payment Result Order Service-->>API Gateway: Order Created API Gateway-->>Client: 201 Created

gRPC - High-performance, language-neutral RPC framework - Protocol buffer serialization - Supports streaming and bidirectional communication - Better performance than REST for internal service communication

Asynchronous Communication

Event-Driven Architecture - Services communicate through events - Loose coupling between services - Better resilience and scalability - Eventual consistency model

sequenceDiagram participant Order Service participant Message Broker participant Payment Service participant Inventory Service participant Notification Service Order Service->>Message Broker: OrderCreated Event Message Broker->>Payment Service: Process Payment Message Broker->>Inventory Service: Reserve Items Payment Service->>Message Broker: PaymentProcessed Event Inventory Service->>Message Broker: ItemsReserved Event Message Broker->>Notification Service: Send Confirmation

Message Patterns

1. **Publish-Subscribe (Pub/Sub)**
2. One-to-many communication
3. Publishers send events to topics
4. Multiple subscribers can consume events
5. Decoupled producers and consumers
6. **Message Queues**
7. Point-to-point communication
8. Guaranteed delivery and ordering
9. Load balancing across consumers
10. Suitable for work distribution
11. **Event Sourcing**
12. Store events as the source of truth
13. Rebuild state from event history
14. Audit trail and temporal queries
15. Complex but powerful pattern

Async Communication Benefits - **Resilience**: Services can continue operating if others are down - **Scalability**: Handle varying loads independently - **Flexibility**: Easy to add new consumers without changing producers - **Performance**: Non-blocking operations improve throughput

Communication Anti-Patterns to Avoid

- **Chatty Interfaces**: Too many fine-grained service calls
- **Shared Databases**: Multiple services accessing the same database
- **Distributed Transactions**: Two-phase commits across services
- **Synchronous Chains**: Long chains of synchronous calls

Transaction Management

Managing transactions in a distributed microservices environment is challenging. Traditional monolithic applications often rely on ACID transactions spanning multiple operations. In microservices, services typically own their data, making distributed transactions impractical. Instead, patterns such as **eventual consistency**, **sagas**, and **compensating transactions** are used:

- **Sagas**: Break a transaction into a series of local transactions, coordinated through events or commands. If a step fails, compensating actions are triggered to maintain consistency.
- **Eventual Consistency**: Accept that data may be temporarily inconsistent, but will become consistent over time as services synchronize via events.
- **Idempotency**: Ensure that repeated operations produce the same result, which is crucial for reliable message processing.

Saga Pattern Example

graph LR A[Order Created] --> B[Reserve Inventory] B --> C[Process Payment] C --> D[Ship Order] D --> E[Order Complete] B -.->|Failure| F[Cancel Order] C -.->|Failure| G[Release Inventory] D -.->|Failure| H[Refund Payment]

Best Practices for Microservices Communication

1. **Design for Failure**: Implement circuit breakers, retries, and timeouts
2. **Use Async When Possible**: Prefer event-driven communication for better resilience
3. **Implement Idempotency**: Ensure operations can be safely retried
4. **Monitor and Trace**: Use distributed tracing to understand request flows

5. **Version APIs**: Plan for backward compatibility and gradual rollouts
6. **Secure Communication**: Use mTLS, API keys, and OAuth for service authentication

Further Reading

- Red Hat Developer: Microservices Architecture
- Red Hat Developer: Bounded Contexts in Microservices
- Red Hat Developer: Distributed Data Management Patterns
- Martin Fowler: Microservices
- Microservices.io: Patterns
- Event-Driven Architecture Patterns
- Saga Pattern

By understanding bounded contexts, communication patterns, and distributed transaction management, you can design robust, scalable microservices that are well-suited for containerized and cloud-native environments.

4.4 Clean Architecture: Building Maintainable and Adaptable Software

4.4.1 Overview

Clean Architecture represents a software design approach that prioritizes long-term maintainability, testability, and adaptability through systematic separation of concerns. This architectural pattern organizes code into distinct layers with clearly defined responsibilities and boundaries, enabling development teams to build applications that remain flexible and maintainable as requirements evolve.

The architecture's principles support modern development practices by isolating business logic from framework dependencies, database implementations, and user interface concerns. This separation enables teams to make technology decisions independently while preserving the core business value embedded in application logic.

4.4.2 Architecture Principles

Separation of Concerns

Clean Architecture implements strict separation between different aspects of application functionality:

- **Business Logic Isolation**: Core business rules remain independent of external frameworks and technologies
- **Infrastructure Independence**: Database, web frameworks, and external services operate as replaceable components
- **User Interface Decoupling**: Presentation logic separates from business logic, enabling multiple interface implementations
- **Cross-Cutting Concerns**: Logging, security, and monitoring integrate without affecting core business operations

Testability Benefits

The architectural approach provides significant advantages for testing practices:

- **Unit Testing**: Business logic testing occurs without database or external service dependencies
- **Integration Testing**: Interface boundaries enable focused testing of component interactions
- **Test Isolation**: Individual layers undergo testing independently, reducing test complexity
- **Mock Implementation**: External dependencies utilize mock implementations during testing phases

Technology Flexibility

Clean Architecture enables technology stack evolution without business logic modifications:

- **Framework Independence**: Application core remains unaffected by web framework changes
- **Database Agnosticism**: Data persistence implementations can change without business logic impact
- **External Service Integration**: Third-party service integrations operate through well-defined interfaces
- **Deployment Environment Adaptation**: Applications deploy across different environments with configuration changes only

4.4.3 Architecture Layers

Dependency Rule

The fundamental principle governing Clean Architecture states that source code dependencies must point inward toward higher-level policies. This rule ensures that:

- Inner layers remain unaware of outer layer implementations
- Business logic maintains independence from framework choices
- Core functionality operates without external system dependencies
- Changes in outer layers do not affect inner layer stability

Layer Structure

Clean Architecture organizes code into four concentric layers, each with distinct responsibilities:

ENTITIES (ENTERPRISE BUSINESS RULES)

The innermost layer contains enterprise-wide business rules and domain models:

- **Domain Objects**: Core business entities representing fundamental business concepts
- **Business Rules**: Enterprise-wide policies and constraints that transcend individual applications
- **Value Objects**: Immutable objects that describe domain characteristics
- **Domain Services**: Operations that don't naturally belong to specific entities

USE CASES (APPLICATION BUSINESS RULES)

The application layer orchestrates business logic for specific use cases:

- **Application Services**: Coordinate domain objects to fulfill business use cases
- **Command Handlers**: Process commands that modify application state
- **Query Handlers**: Execute queries to retrieve application data
- **Business Workflows**: Implement complex business processes involving multiple domain objects

INTERFACE ADAPTERS

The interface layer converts data between use cases and external systems:

- **Controllers**: Handle HTTP requests and coordinate with application services
- **Presenters**: Format data for specific user interface requirements
- **Repository Interfaces**: Define contracts for data persistence operations
- **Gateway Interfaces**: Abstract external service integrations

FRAMEWORKS AND DRIVERS

The outermost layer contains implementation details and external interfaces:

- **Web Frameworks**: ASP.NET Core controllers and middleware
- **Database Systems**: Entity Framework implementations and data access
- **External APIs**: Third-party service integrations and client libraries
- **User Interfaces**: React Native components and mobile platform interfaces

Dependency Inversion

Clean Architecture implements dependency inversion through interface definitions:

- **Interface Definition**: Inner layers define contracts for required external services
- **Implementation Injection**: Outer layers provide concrete implementations of defined interfaces
- **Runtime Binding**: Dependency injection containers resolve interface implementations at runtime
- **Testing Support**: Mock implementations facilitate isolated testing of business logic

4.4.4 Clean Architecture Visual Model

The following diagram illustrates the four layers of Clean Architecture and demonstrates how the Dependency Rule enforces inward-pointing dependencies:

graph TD subgraph "Frameworks & Drivers" A[UI, Web, Database, Devices] end subgraph "Interface Adapters" B[Controllers, Presenters, Gateways] end subgraph "Application Business Rules" C[Use Cases] end subgraph "Enterprise Business Rules" D[Entities] end A --> B B --> C C --> D style D fill:#e1f5fe style C fill:#f3e5f5 style B fill:#e8f5e8 style A fill:#fff3e0

4.4.5 Prerequisites

Development Environment

- Understanding of object-oriented programming principles
- Familiarity with dependency injection concepts
- Experience with .NET Core or React Native development
- Knowledge of containerization using Podman
- Access to Red Hat Universal Base Images

Technical Requirements

- .NET 6.0+ SDK for backend development
- Node.js 18+ for React Native development
- Podman installed and configured
- Visual Studio Code or similar development environment
- Understanding of unit testing frameworks (xUnit, Jest)

4.4.6 Practical Examples

Example 1: .NET Web API with Clean Architecture

This example demonstrates implementing Clean Architecture in a .NET Web API application for order management.

DOMAIN LAYER (ENTITIES)

```
// Domain/Entities/Order.cs
namespace OrderManagement.Domain.Entities
{
    public class Order
    {
        public Guid Id { get; private set; }
        public string CustomerName { get; private set; }
        public DateTime OrderDate { get; private set; }
        public decimal TotalAmount { get; private set; }
        public OrderStatus Status { get; private set; }
        private readonly List<OrderItem> _items = new();
        public IReadOnlyList<OrderItem> Items => _items.AsReadOnly();

        private Order() { } // Required for EF Core

        public Order(string customerName)
        {
            Id = Guid.NewGuid();
            CustomerName = customerName ?? throw new ArgumentNullException(nameof(customerName));
            OrderDate = DateTime.UtcNow;
            Status = OrderStatus.Pending;
            TotalAmount = 0;
        }

        public void AddItem(string productName, decimal price, int quantity)
        {
            if (Status != OrderStatus.Pending)
                throw new InvalidOperationException("Cannot modify confirmed order");

            var item = new OrderItem(productName, price, quantity);
            _items.Add(item);
            RecalculateTotal();
        }

        public void ConfirmOrder()
        {
            if (!_items.Any())
                throw new InvalidOperationException("Cannot confirm empty order");

            Status = OrderStatus.Confirmed;
        }

        private void RecalculateTotal()
        {
            TotalAmount = _items.Sum(item => item.TotalPrice);
        }
    }

    public enum OrderStatus
```

```
    {
        Pending,
        Confirmed,
        Shipped,
        Delivered
    }
}
```

APPLICATION LAYER (USE CASES)

```
// Application/UseCases/CreateOrderUseCase.cs
namespace OrderManagement.Application.UseCases
{
    public interface ICreateOrderUseCase
    {
        Task<CreateOrderResponse> ExecuteAsync(CreateOrderRequest request);
    }

    public class CreateOrderUseCase : ICreateOrderUseCase
    {
        private readonly IOrderRepository _orderRepository;
        private readonly ILogger<CreateOrderUseCase> _logger;

        public CreateOrderUseCase(IOrderRepository orderRepository, ILogger<CreateOrderUseCase> logger)
        {
            _orderRepository = orderRepository ?? throw new ArgumentNullException(nameof(orderRepository));
            _logger = logger ?? throw new ArgumentNullException(nameof(logger));
        }

        public async Task<CreateOrderResponse> ExecuteAsync(CreateOrderRequest request)
        {
            try
            {
                var order = new Order(request.CustomerName);

                foreach (var item in request.Items)
                {
                    order.AddItem(item.ProductName, item.Price, item.Quantity);
                }

                await _orderRepository.SaveAsync(order);

                _logger.LogInformation("Order created successfully: {OrderId}", order.Id);

                return new CreateOrderResponse
                {
                    OrderId = order.Id,
                    TotalAmount = order.TotalAmount,
                    Status = order.Status.ToString()
                };
            }
            catch (Exception ex)
            {
                _logger.LogError(ex, "Failed to create order for customer: {CustomerName}", request.CustomerName);
                throw;
            }
        }
    }
}
```

INTERFACE ADAPTERS (CONTROLLERS)

```
// Infrastructure/Web/Controllers/OrdersController.cs
namespace OrderManagement.Infrastructure.Web.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class OrdersController : ControllerBase
    {
        private readonly ICreateOrderUseCase _createOrderUseCase;
        private readonly ILogger<OrdersController> _logger;

        public OrdersController(ICreateOrderUseCase createOrderUseCase, ILogger<OrdersController> logger)
        {
            _createOrderUseCase = createOrderUseCase ?? throw new ArgumentNullException(nameof(createOrderUseCase));
            _logger = logger ?? throw new ArgumentNullException(nameof(logger));
        }

        [HttpPost]
        public async Task<ActionResult<CreateOrderResponse>> CreateOrder([FromBody] CreateOrderRequest request)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }

            try
            {
                var response = await _createOrderUseCase.ExecuteAsync(request);
                return CreatedAtAction(nameof(GetOrder), new { id = response.OrderId }, response);
            }
```

```
            catch (ArgumentException ex)
            {
                _logger.LogWarning(ex, "Invalid order creation request");
                return BadRequest(ex.Message);
            }
            catch (Exception ex)
            {
                _logger.LogError(ex, "Unexpected error creating order");
                return StatusCode(500, "An error occurred while processing the request");
            }
        }

        [HttpGet("{id}")]
        public async Task<ActionResult<OrderDto>> GetOrder(Guid id)
        {
            // Implementation for retrieving orders
            return NotFound();
        }
    }
}
```

CONTAINER CONFIGURATION

```
FROM registry.access.redhat.com/ubi8/dotnet-60 AS build
WORKDIR /src

# Copy project files and restore dependencies
COPY ["OrderManagement.Api/OrderManagement.Api.csproj", "OrderManagement.Api/"]
COPY ["OrderManagement.Application/OrderManagement.Application.csproj", "OrderManagement.Application/"]
COPY ["OrderManagement.Domain/OrderManagement.Domain.csproj", "OrderManagement.Domain/"]
COPY ["OrderManagement.Infrastructure/OrderManagement.Infrastructure.csproj", "OrderManagement.Infrastructure/"]

RUN dotnet restore "OrderManagement.Api/OrderManagement.Api.csproj"

# Copy source code and build
COPY . .
WORKDIR "/src/OrderManagement.Api"
RUN dotnet build -c Release -o /app/build

FROM build AS publish
RUN dotnet publish -c Release -o /app/publish

# Create runtime image
FROM registry.access.redhat.com/ubi8/dotnet-60-runtime AS runtime
WORKDIR /app

# Create non-root user for security
RUN groupadd -r appuser && useradd -r -g appuser appuser
COPY --from=publish --chown=appuser:appuser /app/publish .
USER appuser

EXPOSE 5000
ENV ASPNETCORE_URLS=http://+:5000
ENTRYPOINT ["dotnet", "OrderManagement.Api.dll"]
```

Example 2: React Native Mobile App with Clean Architecture

This example demonstrates implementing Clean Architecture in a React Native application for the same order management system.

DOMAIN ENTITIES (TYPESCRIPT)

```
// src/domain/entities/Order.ts
export interface OrderItem {
  productName: string;
  price: number;
  quantity: number;
  totalPrice: number;
}

export enum OrderStatus {
  PENDING = 'pending',
  CONFIRMED = 'confirmed',
  SHIPPED = 'shipped',
  DELIVERED = 'delivered'
}

export class Order {
  constructor(
    public readonly id: string,
    public readonly customerName: string,
    public readonly orderDate: Date,
    public readonly items: OrderItem[],
    public readonly status: OrderStatus,
    public readonly totalAmount: number
  ) {}

  static create(customerName: string, items: OrderItem[]): Order {
    const totalAmount = items.reduce((sum, item) => sum + item.totalPrice, 0);
```

```
    return new Order(
      generateId(),
      customerName,
      new Date(),
      items,
      OrderStatus.PENDING,
      totalAmount
    );
  }

  canBeModified(): boolean {
    return this.status === OrderStatus.PENDING;
  }

  isReadyForConfirmation(): boolean {
    return this.items.length > 0 && this.status === OrderStatus.PENDING;
  }
}

function generateId(): string {
  return Math.random().toString(36).substring(2) + Date.now().toString(36);
}
```

APPLICATION USE CASES (TYPESCRIPT)

```
// src/application/usecases/CreateOrderUseCase.ts
export interface CreateOrderRequest {
  customerName: string;
  items: Array<{
    productName: string;
    price: number;
    quantity: number;
  }>;
}

export interface CreateOrderResponse {
  orderId: string;
  totalAmount: number;
  status: string;
}

export interface IOrderRepository {
  save(order: Order): Promise<void>;
  findById(id: string): Promise<Order | null>;
}

export class CreateOrderUseCase {
  constructor(
    private orderRepository: IOrderRepository,
    private logger: ILogger
  ) {}

  async execute(request: CreateOrderRequest): Promise<CreateOrderResponse> {
    try {
      const orderItems: OrderItem[] = request.items.map(item => ({
        productName: item.productName,
        price: item.price,
        quantity: item.quantity,
        totalPrice: item.price * item.quantity
      }));

      const order = Order.create(request.customerName, orderItems);

      if (!order.isReadyForConfirmation()) {
        throw new Error('Order cannot be created without items');
      }

      await this.orderRepository.save(order);

      this.logger.info(`Order created successfully: ${order.id}`);

      return {
        orderId: order.id,
        totalAmount: order.totalAmount,
        status: order.status
      };
    } catch (error) {
      this.logger.error('Failed to create order', error);
      throw error;
    }
  }
}
```

INTERFACE ADAPTERS (PRESENTERS AND CONTROLLERS)

```
// src/presentation/components/CreateOrderScreen.tsx
import React, { useState } from 'react';
import { View, Text, TextInput, TouchableOpacity, Alert, StyleSheet } from 'react-native';
import { CreateOrderUseCase, CreateOrderRequest } from '../../application/usecases/CreateOrderUseCase';

interface CreateOrderScreenProps {
```

```
  createOrderUseCase: CreateOrderUseCase;
  onOrderCreated: (orderId: string) => void;
}

export const CreateOrderScreen: React.FC<CreateOrderScreenProps> = ({
  createOrderUseCase,
  onOrderCreated
}) => {
  const [customerName, setCustomerName] = useState('');
  const [productName, setProductName] = useState('');
  const [price, setPrice] = useState('');
  const [quantity, setQuantity] = useState('');
  const [loading, setLoading] = useState(false);

  const handleCreateOrder = async () => {
    if (!customerName.trim() || !productName.trim() || !price || !quantity) {
      Alert.alert('Error', 'Please fill in all fields');
      return;
    }

    const request: CreateOrderRequest = {
      customerName: customerName.trim(),
      items: [{
        productName: productName.trim(),
        price: parseFloat(price),
        quantity: parseInt(quantity, 10)
      }]
    };

    setLoading(true);

    try {
      const response = await createOrderUseCase.execute(request);
      Alert.alert(
        'Success',
        `Order created successfully!\nTotal: $${response.totalAmount.toFixed(2)}`,
        [{ text: 'OK', onPress: () => onOrderCreated(response.orderId) }]
      );

      // Reset form
      setCustomerName('');
      setProductName('');
      setPrice('');
      setQuantity('');
    } catch (error) {
      Alert.alert('Error', 'Failed to create order. Please try again.');
    } finally {
      setLoading(false);
    }
  };

  return (
    <View style={styles.container}>
      <Text style={styles.title}>Create New Order</Text>

      <TextInput
        style={styles.input}
        placeholder="Customer Name"
        value={customerName}
        onChangeText={setCustomerName}
        editable={!loading}
      />

      <TextInput
        style={styles.input}
        placeholder="Product Name"
        value={productName}
        onChangeText={setProductName}
        editable={!loading}
      />

      <TextInput
        style={styles.input}
        placeholder="Price"
        value={price}
        onChangeText={setPrice}
        keyboardType="decimal-pad"
        editable={!loading}
      />

      <TextInput
        style={styles.input}
        placeholder="Quantity"
        value={quantity}
        onChangeText={setQuantity}
        keyboardType="number-pad"
        editable={!loading}
      />

      <TouchableOpacity
        style={[styles.button, loading && styles.buttonDisabled]}
        onPress={handleCreateOrder}
        disabled={loading}
      >
```

```
        <Text style={styles.buttonText}>
          {loading ? 'Creating...' : 'Create Order'}
        </Text>
      </TouchableOpacity>
    </View>
  );
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    padding: 20,
    backgroundColor: '#f5f5f5'
  },
  title: {
    fontSize: 24,
    fontWeight: 'bold',
    marginBottom: 20,
    textAlign: 'center'
  },
  input: {
    backgroundColor: 'white',
    padding: 15,
    marginBottom: 15,
    borderRadius: 8,
    borderWidth: 1,
    borderColor: '#ddd'
  },
  button: {
    backgroundColor: '#007AFF',
    padding: 15,
    borderRadius: 8,
    alignItems: 'center',
    marginTop: 10
  },
  buttonDisabled: {
    backgroundColor: '#ccc'
  },
  buttonText: {
    color: 'white',
    fontSize: 16,
    fontWeight: '600'
  }
});
```

CONTAINER CONFIGURATION FOR REACT NATIVE

```
FROM registry.access.redhat.com/ubi8/nodejs-18

# Set working directory
WORKDIR /app

# Install React Native CLI and dependencies
USER root
RUN npm install -g @react-native-community/cli react-native-debugger

# Create application user
RUN groupadd -r developer && useradd -r -g developer developer
RUN chown -R developer:developer /app
USER developer

# Copy package files
COPY --chown=developer:developer package*.json ./
RUN npm ci --only=production

# Copy application source
COPY --chown=developer:developer . .

# Expose Metro bundler port
EXPOSE 8081

# Health check
HEALTHCHECK --interval=30s --timeout=10s --start-period=60s --retries=3 \
  CMD curl -f http://localhost:8081/status || exit 1

# Start Metro bundler
CMD ["npm", "start"]
```

4.4.7 Common Pitfalls

Dependency Direction Violations

Problem: Accidentally creating dependencies from inner layers to outer layers.

Solution: Always define interfaces in inner layers and implement them in outer layers:

```

```

```
// Wrong: Use case depends on concrete repository
public class CreateOrderUseCase
{
    private readonly SqlOrderRepository _repository; // Dependency on outer layer
}

// Correct: Use case depends on interface
public class CreateOrderUseCase
{
    private readonly IOrderRepository _repository; // Dependency on inner layer interface
}
```

Anemic Domain Models

Problem: Domain entities that only contain data without behavior.

Solution: Implement rich domain models with business logic:

```
// Wrong: Anemic model
public class Order
{
    public Guid Id { get; set; }
    public string CustomerName { get; set; }
    public decimal Total { get; set; }
}

// Correct: Rich domain model
public class Order
{
    public void AddItem(OrderItem item)
    {
        ValidateBusinessRules(item);
        _items.Add(item);
        RecalculateTotal();
    }
}
```

Over-Engineering

Problem: Creating unnecessary abstractions for simple operations.

Solution: Apply Clean Architecture principles proportionally to application complexity.

4.4.8 Best Practices

Domain Layer Guidelines

1. **Business Logic Encapsulation**: Keep all business rules within domain entities and services
2. **Framework Independence**: Avoid framework dependencies in domain models
3. **Validation Implementation**: Implement business validation within domain objects
4. **Immutability**: Use immutable value objects where appropriate

Application Layer Guidelines

1. **Single Responsibility**: Each use case should handle one specific business operation
2. **Dependency Injection**: Use constructor injection for all dependencies
3. **Error Handling**: Implement comprehensive error handling and logging
4. **Transaction Management**: Handle database transactions at the application layer

Testing Strategies

```
// Unit test for domain logic
[Test]
public void Order_AddItem_ShouldRecalculateTotal()
{
    // Arrange
    var order = new Order("John Doe");
    var item = new OrderItem("Product A", 10.00m, 2);
```

```
    // Act
    order.AddItem(item);

    // Assert
    Assert.AreEqual(20.00m, order.TotalAmount);
}

// Integration test for use case
[Test]
public async Task CreateOrderUseCase_ValidRequest_ShouldCreateOrder()
{
    // Arrange
    var mockRepository = new Mock<IOrderRepository>();
    var useCase = new CreateOrderUseCase(mockRepository.Object, Mock.Of<ILogger>());

    var request = new CreateOrderRequest
    {
        CustomerName = "John Doe",
        Items = new[] { new OrderItemRequest { ProductName = "Product A", Price = 10, Quantity = 2 } }
    };

    // Act
    var response = await useCase.ExecuteAsync(request);

    // Assert
    Assert.IsNotNull(response);
    Assert.AreEqual(20.00m, response.TotalAmount);
    mockRepository.Verify(r => r.SaveAsync(It.IsAny<Order>()), Times.Once);
}
```

4.4.9 Tools and Resources

Development Tools

- **IDEs**: Visual Studio, Visual Studio Code, JetBrains Rider
- **Testing Frameworks**: xUnit, NUnit (for .NET), Jest, Detox (for React Native)
- **Mocking Libraries**: Moq (for .NET), jest.mock (for React Native)
- **Dependency Injection**: Microsoft.Extensions.DependencyInjection, Autofac

Useful Resources

- Clean Architecture by Robert C. Martin
- .NET Application Architecture Guides
- React Native Best Practices
- Container Best Practices with Podman

4.4.10 Hands-on Exercise

Exercise: Implement a Task Management System

Create a task management application using Clean Architecture principles with both .NET Web API backend and React Native frontend.

Requirements:

1. **Domain Features:**
2. Create, update, and delete tasks
3. Assign tasks to users
4. Set task priorities and due dates
5. Track task completion status
6. **Technical Requirements:**
7. Use Red Hat UBI base images for all containers
8. Implement proper error handling and logging
9. Include comprehensive unit and integration tests
10. Use Podman for containerization
11. Implement proper security practices (non-root users)

Steps:

1. **Design the Domain Model:**
2. Define Task and User entities
3. Implement business rules and validation
4. Create value objects for task priorities and status
5. **Implement Use Cases:**
6. CreateTaskUseCase
7. UpdateTaskUseCase
8. AssignTaskUseCase
9. CompleteTaskUseCase
10. **Build Infrastructure:**
11. Repository implementations
12. Database integration (PostgreSQL)
13. API controllers
14. **Create Mobile Interface:**
15. Task list screen
16. Task creation form
17. Task details view
18. **Containerize Applications:**
19. Create Containerfiles for API and mobile development
20. Set up podman compose for local development
21. Configure proper networking and volumes

Deliverables:

- Complete source code with Clean Architecture implementation
- Unit and integration test suites
- Container configuration files
- Documentation explaining architectural decisions
- Performance analysis and optimization recommendations

4.4.11 Summary

Clean Architecture provides a structured approach to building maintainable, testable, and adaptable software systems. Key benefits include:

- **Maintainability**: Clear separation of concerns reduces complexity and technical debt
- **Testability**: Business logic isolation enables comprehensive automated testing
- **Flexibility**: Framework independence allows technology stack evolution
- **Team Productivity**: Well-defined boundaries improve developer understanding and collaboration
- **Long-term Value**: Architecture investment pays dividends throughout application lifecycle

By implementing Clean Architecture principles with modern technologies like .NET, React Native, and containerization using Podman and Red Hat UBI images, development teams can build robust applications that adapt to changing business requirements while maintaining code quality and developer velocity.

4.5 Inner Loop Development

The "Inner Loop" refers to the iterative process a developer goes through to write, build, and debug code before sharing it with the team. A fast and efficient inner loop is crucial for developer productivity and satisfaction, as it allows for rapid feedback and iteration.

4.5.1 IDE-Independent Local Development Environments

To ensure consistency and reduce the "it works on my machine" problem, it's essential to have a development environment that is consistent across the team, regardless of individual IDE preferences. This is where containerization technologies like Docker and Podman come in, and Visual Studio Code's development containers feature provides a seamless way to leverage them.

4.5.2 VS Code Dev Containers

VS Code's Dev Containers feature lets you use containers as a full-featured development environment. It allows you to define your development environment as code, ensuring that everyone on the team has the same tools and dependencies.

Using an Explicit Compose File

For more complex applications that require multiple services (e.g., a backend API, a database, a messaging queue), you can use a Docker Compose file to define the multi-service environment.

You can configure your dev container to use a Compose file by creating a `.devcontainer` directory in your project root and adding a `devcontainer.json` file that points to your `docker-compose.yml`.

Example `devcontainer.json`:

```
{
  "name": "My Project Dev Container",
  "dockerComposeFile": "../docker-compose.yml",
  "service": "app",
  "workspaceFolder": "/workspace",
  "extensions": [
    "ms-azuretools.vscode-docker",
    "dbaeumer.vscode-eslint"
  ],
  "settings": {
    "terminal.integrated.shell.linux": "/bin/bash"
  },
  "forwardPorts": [3000, 5432],
  "postCreateCommand": "npm install"
}
```

In this example: - `dockerComposeFile` points to the `compose.yml` file. - `service` specifies which service in the compose file to use as the development container. - `workspaceFolder` sets the default directory to open in VS Code. - `extensions` lists the VS Code extensions to install in the dev container. - `settings` allows you to configure VS Code settings for the containerized environment. - `forwardPorts` automatically forwards ports from the container to the host. - `postCreateCommand` runs a command after the container is created.

Example `compose.yml`:

```
version: '3.8'
services:
  app:
    build:
      context: .
      Containerfile: Containerfile
    volumes:
      - .:/workspace:cached
    ports:
      - "3000:3000"
    depends_on:
      - db
  db:
    image: postgres:13
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: mydatabase
    ports:
      - "5432:5432"
```

This setup creates a consistent, reproducible, and isolated development environment for every developer on the team, streamlining the inner loop and improving overall development velocity.

4.5.3 Key Benefits of this Approach

Source Code Synchronization with Volume Mounting

The `volumes` key in the `compose.yml` file (e.g., `- .:/workspace:cached`) is crucial for an efficient inner loop. This mounts your local source code directory into the container. Any changes you make to your code on your local machine are instantly reflected inside the container, and vice-versa. This allows you to use your favorite local editor or IDE while the code executes within the fully provisioned containerized environment, providing a seamless development experience.

Toolchain and Dependency Isolation

By defining the development environment in a `Containerfile` or `Containerfile` and `docker-compose.yml`, you are codifying the exact tools, runtimes, and dependencies required for the project. This means developers don't need to manually install and configure these tools on their local machines. The container provides a sandboxed environment with everything needed to build, run, and debug the application. This avoids conflicts with other projects' dependencies and ensures that every team member is working with the exact same toolchain, eliminating "works on my machine" issues.

4.5.4 Development vs. Production Containers

It is important to understand that a development container is not the same as a production container. They serve different purposes and are built differently.

Feature	Development Container	Production Container
Purpose	Provide a consistent and rich environment for writing, debugging, and testing code.	Run the application in a secure, stable, and efficient manner.
Tooling	Includes compilers, debuggers, linters, and other development tools.	Contains only the minimal dependencies required to run the application. No development tools.
Source Code	Mounts the source code from the local filesystem to allow for real-time editing.	The compiled application code is copied into the container image. No source code is present.
Image Size	Larger image size due to the inclusion of development tools and dependencies.	Optimized for a small image size to reduce storage and network overhead.
Security	May run as a non-root user for better security, but the focus is on developer convenience.	Hardened for security. Runs with the minimum required privileges. Minimal attack surface.
Configuration	May contain development-specific configurations, such as mock services or relaxed security settings.	Uses production-ready configurations, including secrets management and robust logging.

A common practice is to use a multi-stage `Containerfile` to build both development and production images from the same source. The development stage includes all the build tools and dependencies, while the production stage copies only the compiled application from the development stage into a minimal base image.

4.6 Overview

In addition to functional requirements—which specify what a system should do—software systems must also meet a set of **non-functional requirements (NFRs)**. These requirements define how a system should behave and set criteria for evaluating the operation of a system, rather than specific behaviors or features.

Non-functional requirements are critical for ensuring that applications are reliable, scalable, secure, and maintainable. They often influence architectural decisions and can have a significant impact on user satisfaction and system success.

4.6.1 What Are Non-Functional Requirements?

Non-functional requirements describe the quality attributes, system properties, and constraints that a solution must have. They answer questions such as:

- How fast should the system respond? (Performance)
- How many users should it support? (Scalability)
- How often can it be unavailable? (Availability)
- How secure must it be? (Security)
- How easy is it to maintain or extend? (Maintainability)
- How well does it recover from failures? (Reliability)
- How easy is it to use? (Usability)
- How portable is it across environments? (Portability)
- How well does it support monitoring and troubleshooting? (Observability)

4.6.2 Common Categories of NFRs

Category	Description
Performance	Response time, throughput, latency, resource usage
Scalability	Ability to handle increased load or users
Availability	Uptime requirements, fault tolerance, disaster recovery
Reliability	Consistency of correct operation over time
Security	Authentication, authorization, data protection, compliance
Maintainability	Ease of updates, bug fixes, and enhancements
Usability	User experience, accessibility, learnability
Portability	Ability to run on different platforms or environments
Observability	Logging, monitoring, tracing, and diagnostics
Compliance	Adherence to legal, regulatory, or industry standards

4.6.3 Why NFRs Matter

Neglecting non-functional requirements can lead to systems that technically work, but fail to meet user or business expectations. For example, an application may provide all required features but be too slow, unreliable, or difficult to maintain. NFRs help ensure that the system is robust, efficient, and ready for real-world use.

4.6.4 NFRs in Modern Application Development

When building cloud-native, containerized, or microservices-based applications, NFRs become even more important. For example:

- **Performance and scalability** must be considered for distributed workloads.
- **Security** is critical in multi-tenant and public cloud environments.
- **Observability** enables effective monitoring and troubleshooting in dynamic, ephemeral infrastructure.
- **Availability and reliability** are essential for always-on services.

4.6.5 Best Practices

- **Define NFRs early:** Capture them alongside functional requirements.
- **Make NFRs measurable:** Use specific, testable criteria (e.g., "99.9% uptime," "response time < 200ms").
- **Prioritize NFRs:** Not all NFRs are equally important—focus on those most critical to your stakeholders.
- **Validate and test:** Use automated tests, monitoring, and reviews to ensure NFRs are met.
- **Iterate:** Revisit NFRs as the system evolves and as usage patterns change.

4.6.6 Further Reading

- Red Hat: Non-functional requirements for cloud-native applications
- OWASP: Security Requirements
- Martin Fowler: Non-Functional Requirements

By carefully considering and implementing non-functional requirements, you can build software that not only works, but excels in real-world environments.

5. 4. Agile Development & Testing

5.1 BDD and TDD Practices

5.1.1 BDD and TDD (Brief Introduction)

5.1.2 What is TDD?

Test-Driven Development (TDD) is a practice where you write a failing test first, implement the minimum code to make it pass, and then refactor. The rhythm is Red → Green → Refactor. TDD improves design, keeps code testable, and gives fast feedback.

5.1.3 What is BDD?

Behavior-Driven Development (BDD) builds on TDD by describing system behavior in business-friendly language. It aligns stakeholders and developers around shared, executable examples of how the system should behave.

5.1.4 What is Gherkin?

Gherkin is a plain-text syntax used in BDD to express behaviors via Given–When–Then steps. It is readable by non-technical stakeholders and automatable by test frameworks.

Small Example (Gherkin)

```
Feature: Shopping cart totals
  As a shopper
  I want my cart total to include tax
  So that I know what I will pay

  Scenario: Calculate total with tax
    Given my cart has an item priced $100
    And the sales tax rate is 10%
    When I view my cart total
    Then the total should be $110
```

5.1.5 When to Use

- Use TDD to guide low-level design and ensure units are correct.
- Use BDD (with Gherkin) to capture behaviors that matter to users and the business.

5.1.6 Learn More

- Requirement-centric development and how BDD fits: Requirement-Centric Development
- BDD concepts and Gherkin: `https://cucumber.io/docs/bdd/`
- TDD overview: `https://martinfowler.com/bliki/TestDrivenDevelopment.html`

This page is intentionally concise and introductory. Refer to the links above for deeper, hands-on guidance.

5.1.7 Quick Examples

React Native (TDD)

```
// sum.test.ts
import { sum } from './sum';

test('sum adds numbers', () => {
  expect(sum(2, 3)).toBe(5);
});
```

```
// sum.ts
export const sum = (a: number, b: number) => a + b;
```

.NET (TDD)

```
// MathTests.cs
using Xunit;

public class MathTests
{
    [Fact]
    public void Sum_AddsNumbers()
    {
        Assert.Equal(5, MathUtil.Sum(2, 3));
    }
}
```

```
public static class MathUtil
{
    public static int Sum(int a, int b) => a + b;
}
```

5.2 Local Development with podman compose

5.2.1 Overview

Local development environments are crucial for developer productivity and application quality. Using podman compose , developers can create consistent, reproducible development environments that closely mirror production while remaining lightweight and fast to start. This guide demonstrates how to set up and manage local development environments for .NET and React Native applications using podman compose with Red Hat UBI images.

podman compose provides the same functionality as Docker Compose but with enhanced security through rootless containers, making it ideal for enterprise development environments where security is paramount.

5.2.2 Key Concepts

podman compose Benefits

- **Rootless Containers**: Enhanced security without requiring root privileges
- **Development Consistency**: Same environment across all developer machines
- **Production Parity**: Mirror production architecture locally
- **Resource Efficiency**: Lightweight containers start quickly
- **Service Isolation**: Each service runs independently
- **Easy Debugging**: Individual service access and log monitoring

Development vs Production

Aspect	Development	Production
Volume Mounts	Source code hot reload	Built application artifacts
Environment Variables	Development settings	Production configuration
Networking	Direct port access	Load balancer/ingress
Persistence	Temporary volumes	Persistent storage
Monitoring	Basic logging	Full observability stack
Security	Simplified for debugging	Hardened security policies

Container Orchestration Patterns

- **Multi-service Applications**: Frontend, backend, and database coordination
- **Service Dependencies**: Proper startup ordering and health checks
- **Development Overrides**: Different configurations for local development
- **Hot Reloading**: Live code updates without container rebuilds
- **Database Management**: Consistent data seeding and migrations

5.2.3 Prerequisites

- Podman and podman compose installed and configured
- Understanding of containerization concepts
- Basic knowledge of YAML configuration
- Familiarity with .NET and React Native development

- Understanding of networking and service communication

Installation Requirements

```
# Install Podman and podman compose
# On RHEL/CentOS/Fedora
sudo dnf install podman podman-compose

# On macOS
brew install podman
pip3 install podman-compose

# Verify installation
podman --version
podman compose --version
```

5.2.4 Practical Examples

Example 1: Airline Booking Development Environment

Let's create a comprehensive development environment for an airline booking application with multiple services.

STEP 1: PROJECT STRUCTURE

```
airline-dev/
├── compose.yml
├── podman-compose.override.yml
├── .env
├── api/
│   ├── Containerfile.dev
│   ├── UserService/
│   ├── FlightService/
│   ├── BookingService/
│   └── PaymentService/
├── web/
│   ├── Containerfile.dev
│   └── src/
├── mobile/
│   ├── Containerfile.dev
│   └── src/
├── database/
│   ├── init-scripts/
│   └── seed-data/
└── infrastructure/
    ├── nginx/
    ├── monitoring/
    └── scripts/
```

STEP 2: MAIN COMPOSE CONFIGURATION

```
# compose.yml
version: '3.8'

services:
  # Reverse Proxy and Load Balancer
  nginx:
    image: registry.access.redhat.com/ubi8/nginx-120
    container_name: airline-nginx
    ports:
      - "80:8080"
      - "443:8443"
    volumes:
      - ./infrastructure/nginx/nginx.conf:/etc/nginx/nginx.conf:Z
      - ./infrastructure/nginx/ssl:/etc/nginx/ssl:Z
    depends_on:
      - user-service
      - flight-service
      - booking-service
      - web-app
    networks:
      - app-network
    restart: unless-stopped

  # User Management Service
  user-service:
    build:
      context: ./api/UserService
      Containerfile: Containerfile.dev
    container_name: airline-user-service
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=http://+:8080
      - ConnectionStrings__DefaultConnection=Host=postgres;Database=userdb;Username=userservice;Password=${DB_PASSWORD}
      - JWT__SecretKey=${JWT_SECRET}
```

```
      - JWT__Issuer=ecommerce-dev
      - JWT__Audience=ecommerce-users
      - Redis__ConnectionString=redis:6379
      - Logging__LogLevel__Default=Information
      - Logging__LogLevel__Microsoft=Warning
    ports:
      - "5001:8080"
    depends_on:
      postgres:
        condition: service_healthy
      redis:
        condition: service_healthy
    volumes:
      - ./api/UserService:/app:Z
      - user-service-nuget:/root/.nuget/packages
    networks:
      - app-network
    restart: unless-stopped
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8080/health"]
      interval: 30s
      timeout: 10s
      retries: 3
      start_period: 40s

  # Flight Service
  flight-service:
    build:
      context: ./api/FlightService
      Containerfile: Containerfile.dev
    container_name: airline-flight-service
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=http://+:8080
      - ConnectionStrings__DefaultConnection=Host=postgres;Database=flightdb;Username=flightservice;Password=${DB_PASSWORD}
      - Redis__ConnectionString=redis:6379
      - EventBus__ConnectionString=amqp://guest:guest@rabbitmq:5672
      - Storage__ConnectionString=minio:9000
      - Storage__AccessKey=${MINIO_ACCESS_KEY}
      - Storage__SecretKey=${MINIO_SECRET_KEY}
    ports:
      - "5002:8080"
    depends_on:
      postgres:
        condition: service_healthy
      redis:
        condition: service_healthy
      rabbitmq:
        condition: service_healthy
    volumes:
      - ./api/FlightService:/app:Z
      - flight-service-nuget:/root/.nuget/packages
    networks:
      - app-network
    restart: unless-stopped
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8080/health"]
      interval: 30s
      timeout: 10s
      retries: 3
      start_period: 40s

  # Booking Service
  booking-service:
    build:
      context: ./api/BookingService
      Containerfile: Containerfile.dev
    container_name: airline-booking-service
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=http://+:8080
      - ConnectionStrings__DefaultConnection=Host=postgres;Database=bookingdb;Username=bookingservice;Password=${DB_PASSWORD}
      - Services__UserService__BaseUrl=http://user-service:8080
      - Services__FlightService__BaseUrl=http://flight-service:8080
      - Services__PaymentService__BaseUrl=http://payment-service:8080
      - EventBus__ConnectionString=amqp://guest:guest@rabbitmq:5672
      - Redis__ConnectionString=redis:6379
    ports:
      - "5003:8080"
    depends_on:
      postgres:
        condition: service_healthy
      redis:
        condition: service_healthy
      rabbitmq:
        condition: service_healthy
      user-service:
        condition: service_healthy
      flight-service:
        condition: service_healthy
    volumes:
      - ./api/BookingService:/app:Z
      - booking-service-nuget:/root/.nuget/packages
    networks:
```

```
      - app-network
    restart: unless-stopped
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8080/health"]
      interval: 30s
      timeout: 10s
      retries: 3
      start_period: 40s

  # Payment Processing Service
  payment-service:
    build:
      context: ./api/PaymentService
      Containerfile: Containerfile.dev
    container_name: airline-payment-service
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=http://+:8080
      - ConnectionStrings__DefaultConnection=Host=postgres;Database=paymentdb;Username=paymentservice;Password=${DB_PASSWORD}
      - EventBus__ConnectionString=amqp://guest:guest@rabbitmq:5672
      - PaymentGateway__ApiKey=${PAYMENT_GATEWAY_API_KEY}
      - PaymentGateway__SecretKey=${PAYMENT_GATEWAY_SECRET_KEY}
      - PaymentGateway__Environment=sandbox
    ports:
      - "5004:8080"
    depends_on:
      postgres:
        condition: service_healthy
      rabbitmq:
        condition: service_healthy
    volumes:
      - ./api/PaymentService:/app:Z
      - payment-service-nuget:/root/.nuget/packages
    networks:
      - app-network
    restart: unless-stopped
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8080/health"]
      interval: 30s
      timeout: 10s
      retries: 3
      start_period: 40s

  # Booking Web Application (React)
  web-app:
    build:
      context: ./web
      Containerfile: Containerfile.dev
    container_name: airline-web-app
    environment:
      - NODE_ENV=development
      - REACT_APP_API_BASE_URL=http://localhost/api
      - REACT_APP_WEBSOCKET_URL=ws://localhost/ws
      - CHOKIDAR_USEPOLLING=true
    ports:
      - "3000:3000"
    volumes:
      - ./web:/app:Z
      - web-app-node-modules:/app/node_modules
    networks:
      - app-network
    restart: unless-stopped
    stdin_open: true
    tty: true

  # PostgreSQL Database
  postgres:
    image: registry.access.redhat.com/rhel8/postgresql-13
    container_name: airline-postgres
    environment:
      - POSTGRESQL_USER=postgres
      - POSTGRESQL_PASSWORD=${DB_PASSWORD}
      - POSTGRESQL_DATABASE=postgres
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/pgsql/data
      - ./database/init-scripts:/docker-entrypoint-initdb.d:Z
    networks:
      - app-network
    restart: unless-stopped
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 10s
      timeout: 5s
      retries: 5

  # Redis Cache
  redis:
    image: registry.access.redhat.com/rhel8/redis-6
    container_name: airline-redis
    command: redis-server --appendonly yes --requirepass ${REDIS_PASSWORD}
    ports:
      - "6379:6379"
```

```
    volumes:
      - redis_data:/var/lib/redis/data
    networks:
      - app-network
    restart: unless-stopped
    healthcheck:
      test: ["CMD", "redis-cli", "--raw", "incr", "ping"]
      interval: 10s
      timeout: 3s
      retries: 5

  # RabbitMQ Message Broker
  rabbitmq:
    image: registry.access.redhat.com/ubi8/ubi:latest
    container_name: airline-rabbitmq
    environment:
      - RABBITMQ_DEFAULT_USER=guest
      - RABBITMQ_DEFAULT_PASS=guest
      - RABBITMQ_DEFAULT_VHOST=/
    ports:
      - "5672:5672"
      - "15672:15672"
    volumes:
      - rabbitmq_data:/var/lib/rabbitmq
      - ./infrastructure/rabbitmq/rabbitmq.conf:/etc/rabbitmq/rabbitmq.conf:Z
    networks:
      - app-network
    restart: unless-stopped
    healthcheck:
      test: rabbitmq-diagnostics -q ping
      interval: 30s
      timeout: 10s
      retries: 5

  # MinIO Object Storage
  minio:
    image: quay.io/minio/minio:latest
    container_name: airline-minio
    command: server /data --console-address ":9001"
    environment:
      - MINIO_ROOT_USER=${MINIO_ACCESS_KEY}
      - MINIO_ROOT_PASSWORD=${MINIO_SECRET_KEY}
    ports:
      - "9000:9000"
      - "9001:9001"
    volumes:
      - minio_data:/data
    networks:
      - app-network
    restart: unless-stopped
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:9000/minio/health/live"]
      interval: 30s
      timeout: 20s
      retries: 3

volumes:
  postgres_data:
  redis_data:
  rabbitmq_data:
  minio_data:
  user-service-nuget:
  flight-service-nuget:
  booking-service-nuget:
  payment-service-nuget:
  web-app-node-modules:

networks:
  app-network:
    driver: bridge
    ipam:
      config:
        - subnet: 172.20.0.0/16
```

STEP 3: DEVELOPMENT OVERRIDE CONFIGURATION

```
# podman-compose.override.yml
version: '3.8'

services:
  user-service:
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - Logging__LogLevel__Default=Debug
      - Logging__LogLevel__Microsoft=Information
      - Development__EnableSensitiveDataLogging=true
    volumes:
      - ./api/UserService:/app:Z
      - user-service-nuget:/root/.nuget/packages
    command: >
      bash -c "
        dotnet restore &&
        dotnet watch run --urls http://+:8080
```

```
      "

  flight-service:
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - Logging__LogLevel__Default=Debug
      - Development__EnableSensitiveDataLogging=true
    volumes:
      - ./api/FlightService:/app:Z
      - flight-service-nuget:/root/.nuget/packages
    command: >
      bash -c "
        dotnet restore &&
        dotnet watch run --urls http://+:8080
      "

  booking-service:
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - Logging__LogLevel__Default=Debug
      - Development__EnableSensitiveDataLogging=true
    volumes:
      - ./api/BookingService:/app:Z
      - booking-service-nuget:/root/.nuget/packages
    command: >
      bash -c "
        dotnet restore &&
        dotnet watch run --urls http://+:8080
      "

  payment-service:
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - Logging__LogLevel__Default=Debug
      - Development__EnableSensitiveDataLogging=true
    volumes:
      - ./api/PaymentService:/app:Z
      - payment-service-nuget:/root/.nuget/packages
    command: >
      bash -c "
        dotnet restore &&
        dotnet watch run --urls http://+:8080
      "

  web-app:
    environment:
      - NODE_ENV=development
      - FAST_REFRESH=true
      - CHOKIDAR_USEPOLLING=true
      - WATCHPACK_POLLING=true
    command: npm start
    stdin_open: true
    tty: true

  # Development tools
  pgadmin:
    image: dpage/pgadmin4:latest
    container_name: ecommerce-pgadmin
    environment:
      - PGADMIN_DEFAULT_EMAIL=admin@example.com
      - PGADMIN_DEFAULT_PASSWORD=admin
    ports:
      - "5050:80"
    volumes:
      - pgadmin_data:/var/lib/pgadmin
    networks:
      - app-network
    restart: unless-stopped

  redis-commander:
    image: rediscommander/redis-commander:latest
    container_name: ecommerce-redis-commander
    environment:
      - REDIS_HOSTS=local:redis:6379
      - REDIS_PASSWORD=${REDIS_PASSWORD}
    ports:
      - "8081:8081"
    networks:
      - app-network
    restart: unless-stopped

volumes:
  pgadmin_data:
```

STEP 4: ENVIRONMENT CONFIGURATION

```
# .env
# Database Configuration
DB_PASSWORD=dev_password_123
POSTGRES_DB=ecommerce_dev

# Redis Configuration
REDIS_PASSWORD=redis_dev_password
```

```
# JWT Configuration
JWT_SECRET=your_super_secret_jwt_key_for_development_only

# MinIO Configuration
MINIO_ACCESS_KEY=minioadmin
MINIO_SECRET_KEY=minioadmin123

# Payment Gateway (Sandbox)
PAYMENT_GATEWAY_API_KEY=sandbox_api_key
PAYMENT_GATEWAY_SECRET_KEY=sandbox_secret_key

# Development Settings
ASPNETCORE_ENVIRONMENT=Development
NODE_ENV=development
```

STEP 5: DEVELOPMENT CONTAINERFILES

.NET Service Development Containerfile:

```
# api/UserService/Containerfile.dev
FROM registry.access.redhat.com/ubi8/dotnet-60

# Install development tools
USER root
RUN dnf update -y && \
    dnf install -y curl procps-ng && \
    dnf clean all

# Create app user
RUN groupadd -r appuser && useradd -r -g appuser appuser

# Set working directory
WORKDIR /app

# Copy project files
COPY --chown=appuser:appuser *.csproj ./
RUN dotnet restore

# Copy source code
COPY --chown=appuser:appuser . ./

# Switch to app user
USER appuser

# Expose port
EXPOSE 8080

# Install dotnet tools for hot reload
RUN dotnet tool install --global dotnet-ef
RUN dotnet tool install --global dotnet-watch
ENV PATH="$PATH:/home/appuser/.dotnet/tools"

# Default command for development (can be overridden)
CMD ["dotnet", "watch", "run", "--urls", "http://+:8080"]
```

React Development Containerfile:

```
# web/Containerfile.dev
FROM registry.access.redhat.com/ubi8/nodejs-16

# Install development tools
USER root
RUN dnf update -y && \
    dnf install -y git curl && \
    dnf clean all

# Create app user
RUN groupadd -r appuser && useradd -r -g appuser appuser

# Set working directory
WORKDIR /app

# Copy package files
COPY --chown=appuser:appuser package*.json ./

# Switch to app user
USER appuser

# Install dependencies
RUN npm ci

# Copy source code
COPY --chown=appuser:appuser . ./

# Expose port
EXPOSE 3000

# Set environment for development
ENV NODE_ENV=development
```

```
ENV CHOKIDAR_USEPOLLING=true
ENV FAST_REFRESH=true

# Start development server
CMD ["npm", "start"]
```

Example 2: React Native Development Environment

For React Native development, we need a specialized setup:

```
# mobile-dev/compose.yml
version: '3.8'

services:
  # React Native Metro Bundler
  metro-bundler:
    build:
      context: .
      Containerfile: Containerfile.metro
    container_name: rn-metro-bundler
    environment:
      - NODE_ENV=development
      - REACT_NATIVE_PACKAGER_HOSTNAME=0.0.0.0
    ports:
      - "8081:8081"  # Metro bundler
      - "9090:9090"  # Flipper
    volumes:
      - ./src:/app/src:Z
      - ./assets:/app/assets:Z
      - rn-node-modules:/app/node_modules
    networks:
      - mobile-network
    restart: unless-stopped

  # Android Development Environment
  android-dev:
    build:
      context: .
      Containerfile: Containerfile.android
    container_name: rn-android-dev
    environment:
      - DISPLAY=${DISPLAY}
    volumes:
      - /tmp/.X11-unix:/tmp/.X11-unix:Z
      - ./:/app:Z
      - android-sdk:/opt/android-sdk
      - gradle-cache:/root/.gradle
    networks:
      - mobile-network
    privileged: true
    restart: unless-stopped

  # API Mock Server for Development
  mock-api:
    image: registry.access.redhat.com/ubi8/nodejs-16
    container_name: rn-mock-api
    working_dir: /app
    command: >
      bash -c "
        npm install -g json-server &&
        json-server --watch db.json --host 0.0.0.0 --port 3001
      "
    ports:
      - "3001:3001"
    volumes:
      - ./mock-data:/app:Z
    networks:
      - mobile-network
    restart: unless-stopped

volumes:
  rn-node-modules:
  android-sdk:
  gradle-cache:

networks:
  mobile-network:
    driver: bridge
```

Example 3: Database Management and Seeding

```
-- database/init-scripts/01-create-databases.sql
-- Create separate databases for each service
CREATE DATABASE userdb;
CREATE DATABASE flightdb;
CREATE DATABASE bookingdb;
CREATE DATABASE paymentdb;
```

```
-- Create service users
CREATE USER userservice WITH ENCRYPTED PASSWORD 'dev_password_123';
CREATE USER flightservice WITH ENCRYPTED PASSWORD 'dev_password_123';
CREATE USER bookingservice WITH ENCRYPTED PASSWORD 'dev_password_123';
CREATE USER paymentservice WITH ENCRYPTED PASSWORD 'dev_password_123';

-- Grant permissions
GRANT ALL PRIVILEGES ON DATABASE userdb TO userservice;
GRANT ALL PRIVILEGES ON DATABASE flightdb TO flightservice;
GRANT ALL PRIVILEGES ON DATABASE bookingdb TO bookingservice;
GRANT ALL PRIVILEGES ON DATABASE paymentdb TO paymentservice;
```

```
-- database/init-scripts/02-seed-data.sql
\c flightdb;

-- Sample flight data for development
INSERT INTO flights (id, flight_number, origin, destination, departure_time, arrival_time, base_fare, seats_available, created_at) VALUES
('11111111-1111-1111-1111-111111111111', 'AI101', 'SFO', 'JFK', NOW() + INTERVAL '1 day', NOW() + INTERVAL '1 day 5 hours', 299.99, 120, NOW()),
('33333333-3333-3333-3333-333333333333', 'AI202', 'LAX', 'ORD', NOW() + INTERVAL '2 days', NOW() + INTERVAL '2 days 4 hours', 199.99, 150, NOW()),
('55555555-5555-5555-5555-555555555555', 'AI303', 'SEA', 'BOS', NOW() + INTERVAL '3 days', NOW() + INTERVAL '3 days 5 hours', 249.99, 100, NOW());

\c userdb;

-- Sample user data for development
INSERT INTO users (id, email, first_name, last_name, password_hash, created_at) VALUES
('aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa', 'john.doe@example.com', 'John', 'Doe', '$2a$11$example_hash', NOW()),
('bbbbbbbb-bbbb-bbbb-bbbb-bbbbbbbbbbbb', 'jane.smith@example.com', 'Jane', 'Smith', '$2a$11$example_hash', NOW());
```

5.2.5 Common Pitfalls

1. Volume Mount Issues

Problem: File permission errors or changes not reflecting in containers.

Solution: Use proper SELinux contexts and understand rootless mapping:

```
services:
  api:
    volumes:
      # Correct: Use :Z for SELinux relabeling
      - ./src:/app/src:Z
      # Incorrect: Missing SELinux context
      - ./src:/app/src
```

2. Service Startup Dependencies

Problem: Services starting before dependencies are ready.

Solution: Use proper health checks and depends_on conditions:

```
services:
  api:
    depends_on:
      database:
        condition: service_healthy
      redis:
        condition: service_healthy
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8080/health"]
      interval: 30s
      timeout: 10s
      retries: 3
      start_period: 40s
```

3. Network Connectivity Issues

Problem: Services can't communicate or resolve each other's names.

Solution: Use explicit networks and proper service naming:

```
networks:
  app-network:
    driver: bridge
    ipam:
      config:
        - subnet: 172.20.0.0/16

services:
```

```
  api:
    networks:
      - app-network
    environment:
      - DATABASE_HOST=postgres  # Use service name for internal communication
```

4. Resource Consumption

Problem: Development environment consuming too many system resources.

Solution: Set resource limits and use efficient base images:

```
services:
  api:
    deploy:
      resources:
        limits:
          memory: 512M
          cpus: '0.5'
        reservations:
          memory: 256M
          cpus: '0.25'
```

5.2.6 Best Practices

1. Environment Configuration

```
# Use environment-specific overrides
# compose.yml - Base configuration
# podman-compose.override.yml - Development overrides
# podman-compose.prod.yml - Production configuration

services:
  api:
    environment:
      - ASPNETCORE_ENVIRONMENT=${ENVIRONMENT:-Development}
      - DATABASE_URL=${DATABASE_URL}
      - REDIS_URL=${REDIS_URL}
```

2. Development Scripts

Create helper scripts for common development tasks:

```
#!/bin/bash
# scripts/dev-start.sh

set -e

echo "  Starting development environment..."

# Load environment variables
if [ -f .env ]; then
    export $(grep -v '^#' .env | xargs)
fi

# Start services
podman compose up -d postgres redis rabbitmq

# Wait for databases to be ready
echo "  Waiting for databases..."
sleep 10

# Run database migrations
echo "  Running database migrations..."
podman compose exec user-service dotnet ef database update
podman compose exec product-service dotnet ef database update
podman compose exec order-service dotnet ef database update

# Start application services
echo "  Starting application services..."
podman compose up -d

echo "  Development environment is ready!"
echo "  Web app: http://localhost:3000"
echo "  API docs: http://localhost/swagger"
echo "   Database admin: http://localhost:5050"
echo "  Redis admin: http://localhost:8081"
```

```
#!/bin/bash
# scripts/dev-stop.sh
```

```
echo "  Stopping development environment..."
podman compose down

echo "  Cleaning up..."
podman system prune -f --volumes

echo "  Development environment stopped!"
```

3. Hot Reloading Configuration

```
// Program.cs for .NET services
if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
    app.UseSwagger();
    app.UseSwaggerUI();

    // Enable detailed errors for development
    app.UseDeveloperExceptionPage();

    // Enable CORS for local development
    app.UseCors(builder => builder
        .AllowAnyOrigin()
        .AllowAnyMethod()
        .AllowAnyHeader());
}
```

4. Logging and Debugging

```
services:
  api:
    environment:
      - Logging__LogLevel__Default=Debug
      - Logging__LogLevel__Microsoft=Information
      - Logging__Console__IncludeScopes=true
    volumes:
      - ./logs:/app/logs:Z
```

5. Security in Development

```
# Use secrets for sensitive data even in development
secrets:
  db_password:
    file: ./secrets/db_password.txt
  jwt_secret:
    file: ./secrets/jwt_secret.txt

services:
  api:
    secrets:
      - db_password
      - jwt_secret
    environment:
      - ConnectionStrings__DefaultConnection=Host=postgres;Database=mydb;Username=user;Password_File=/run/secrets/db_password
```

5.2.7 Tools and Resources

Development Tools

- **Podman Desktop**: GUI for container management
- **Visual Studio Code**: With Podman and development container extensions
- **Postman**: API testing and documentation
- **pgAdmin**: PostgreSQL administration
- **Redis Commander**: Redis data browser

Monitoring and Debugging

```
# Add to podman-compose.override.yml for development monitoring
services:
  prometheus:
    image: prom/prometheus:latest
    ports:
      - "9090:9090"
```

```
    volumes:
      - ./monitoring/prometheus-dev.yml:/etc/prometheus/prometheus.yml:Z

  grafana:
    image: grafana/grafana:latest
    ports:
      - "3001:3000"
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=admin
    volumes:
      - ./monitoring/grafana/dashboards:/var/lib/grafana/dashboards:Z
```

Useful Commands

```
# Start all services
podman compose up -d

# View logs for specific service
podman compose logs -f user-service

# Execute commands in running container
podman compose exec user-service bash

# Scale a service
podman compose up -d --scale product-service=2

# Rebuild and restart a service
podman compose up -d --build user-service

# Stop and remove all containers
podman compose down

# Remove volumes (careful with data loss)
podman compose down -v

# View resource usage
podman stats

# Clean up unused resources
podman system prune -f
```

5.2.8 Hands-on Exercise

Exercise: Complete Microservices Development Environment

Create a comprehensive development environment for a microservices-based application.

Requirements:

1. **Application Services:**
2. Authentication service (.NET)
3. Product catalog service (.NET)
4. Order management service (.NET)
5. Notification service (.NET)
6. Web frontend (React)
7. Mobile app development (React Native)
8. **Infrastructure Services:**
9. PostgreSQL database with multiple schemas
10. Redis for caching and sessions
11. RabbitMQ for event-driven communication
12. MinIO for file storage
13. Nginx as reverse proxy
14. **Development Features:**
15. Hot reloading for all services
16. Database seeding with test data
17. Automated database migrations
18. Health checks for all services
19. Centralized logging
20. Development monitoring dashboard
21. **Developer Experience:**
22. One-command startup and shutdown
23. Consistent environment across team
24. Easy debugging and testing
25. Documentation and helper scripts

Deliverables:

1. Complete podman compose configuration
2. Development Containerfiles for each service
3. Database initialization and seeding scripts
4. Development helper scripts
5. Environment configuration management
6. Documentation for onboarding new developers
7. Performance optimization guide

Evaluation Criteria:

- Environment startup time (< 2 minutes)
- Resource efficiency (< 4GB RAM usage)
- Hot reload functionality
- Service communication reliability
- Documentation quality
- Developer onboarding experience

5.2.9 Summary

Local development with podman compose provides a powerful foundation for modern application development. Key takeaways include:

- **Consistent Environments**: podman compose ensures all developers work with identical setups
- **Production Parity**: Local environments mirror production architecture and configuration
- **Enhanced Security**: Rootless containers provide better security isolation
- **Developer Productivity**: Hot reloading and automated setup reduce development friction
- **Service Orchestration**: Multi-service applications can be managed as cohesive units
- **Resource Efficiency**: Lightweight containers minimize system resource usage
- **Debugging Capabilities**: Individual service access and comprehensive logging aid troubleshooting

By implementing these practices with .NET and React Native applications using podman compose and Red Hat UBI images, development teams can achieve faster development cycles, improved code quality, and seamless collaboration. The investment in setting up comprehensive development environments pays dividends in reduced debugging time, consistent behavior across environments, and faster onboarding of new team members.

5.3 Mocking Dependent Systems and APIs

5.3.1 Overview

Modern software applications integrate with multiple external systems including payment processors, authentication services, data providers, and third-party APIs. While these integrations enable rich functionality, they introduce significant challenges for testing, development, and continuous integration processes.

Mocking provides a systematic approach to simulate external system behavior, enabling comprehensive testing without the complexity, cost, and reliability issues associated with live integrations. This guide focuses on practical mocking strategies for .NET and React Native applications using industry-standard tools and containerized approaches.

5.3.2 External Dependency Challenges

Testing Complexity

External system dependencies introduce several obstacles to effective testing:

- **Environment Inconsistency**: External services may behave differently across development, staging, and production environments
- **Network Reliability**: Internet connectivity issues, service outages, and network latency affect test reliability
- **Rate Limiting**: API quotas and throttling mechanisms limit the volume of test requests
- **Financial Costs**: Pay-per-use APIs accumulate costs during extensive testing cycles
- **Data Management**: Live systems may create persistent data that affects subsequent test runs

Development Workflow Impact

External dependencies affect development productivity through:

- **Slow Feedback Cycles**: Network requests significantly increase test execution time
- **Offline Development**: Inability to develop or test without internet connectivity
- **Scenario Coverage**: Difficulty triggering specific edge cases, error conditions, or failure modes
- **Team Coordination**: Shared external resources may create conflicts between development teams

5.3.3 Mocking Strategy Benefits

Test Reliability

Mocking implementations provide predictable behavior that supports reliable testing:

- **Deterministic Responses**: Controlled output enables consistent test results
- **Error Simulation**: Systematic testing of failure scenarios and edge cases
- **Performance Consistency**: Elimination of network variability improves test reliability
- **Isolation**: Individual component testing without external system dependencies

Development Efficiency

Mock implementations enhance development workflows through:

- **Rapid Iteration**: Immediate feedback without network delays
- **Offline Capability**: Development and testing without external system availability
- **Cost Reduction**: Elimination of API usage charges during development
- **Parallel Development**: Teams can work independently of external system availability

5.3.4 Testing Implementation Strategies

Unit Testing with Dependency Injection

Mock individual dependencies at the component level using dependency injection patterns:

Advantages:

- Rapid test execution with minimal overhead
- Complete control over dependency behavior
- Isolated testing of business logic
- Simple setup and configuration

Implementation Focus:

- Interface-based dependency abstraction
- Constructor injection for testability
- Mock framework integration with testing containers

Integration Testing with Service Stubs

Replace complete external systems with lightweight stub implementations:

Advantages:

- End-to-end workflow validation
- Network protocol testing
- Service contract verification
- Multi-component integration scenarios

Implementation Focus:

- HTTP-based stub services
- Message queue simulation
- Database connection mocking
- Authentication service stubs

Contract Testing Implementation

Establish and verify API contracts between service consumers and providers:

Advantages:

- Consumer-driven contract validation
- Breaking change detection
- Team coordination through shared contracts
- Provider behavior verification

Implementation Focus:

- Schema definition and validation
- Contract generation and publishing
- Automated contract testing
- Version compatibility verification

5.3.5 .NET Mocking Implementation

Moq Framework with Dependency Injection

Implement unit testing with Moq and Microsoft.Extensions.DependencyInjection:

```
// Service interface definition
public interface IPaymentService
{
    Task<PaymentResult> ProcessPaymentAsync(PaymentRequest request);
}

// Service implementation under test
public class OrderService
{
    private readonly IPaymentService _paymentService;

    public OrderService(IPaymentService paymentService)
    {
        _paymentService = paymentService;
    }

    public async Task<OrderResult> CreateOrderAsync(CreateOrderRequest request)
    {
        var paymentResult = await _paymentService.ProcessPaymentAsync(
            new PaymentRequest
            {
                Amount = request.Total,
                CardToken = request.PaymentToken
            });

        if (!paymentResult.IsSuccessful)
        {
            return OrderResult.Failed("Payment processing failed");
        }

        return OrderResult.Success(new Order { Id = Guid.NewGuid() });
    }
}

// xUnit test implementation
[Fact]
public async Task CreateOrderAsync_PaymentSuccessful_ReturnsSuccessResult()
{
    // Arrange
    var mockPaymentService = new Mock<IPaymentService>();
    mockPaymentService
        .Setup(x => x.ProcessPaymentAsync(It.IsAny<PaymentRequest>()))
        .ReturnsAsync(new PaymentResult { IsSuccessful = true, TransactionId = "12345" });

    var orderService = new OrderService(mockPaymentService.Object);
    var request = new CreateOrderRequest { Total = 100.00m, PaymentToken = "token123" };

    // Act
    var result = await orderService.CreateOrderAsync(request);

    // Assert
    Assert.True(result.IsSuccess);
    mockPaymentService.Verify(
        x => x.ProcessPaymentAsync(It.Is<PaymentRequest>(p => p.Amount == 100.00m)),
        Times.Once);
}

[Fact]
public async Task CreateOrderAsync_PaymentFailed_ReturnsFailureResult()
{
    // Arrange
    var mockPaymentService = new Mock<IPaymentService>();
    mockPaymentService
        .Setup(x => x.ProcessPaymentAsync(It.IsAny<PaymentRequest>()))
        .ReturnsAsync(new PaymentResult { IsSuccessful = false, ErrorMessage = "Insufficient funds" });

    var orderService = new OrderService(mockPaymentService.Object);
    var request = new CreateOrderRequest { Total = 100.00m, PaymentToken = "token123" };

    // Act
    var result = await orderService.CreateOrderAsync(request);

    // Assert
    Assert.False(result.IsSuccess);
    Assert.Contains("Payment processing failed", result.ErrorMessage);
}
```

WireMock.NET for Integration Testing

Implement HTTP service mocking for integration tests:

```
// Test setup with WireMock server
public class PaymentIntegrationTests : IDisposable
{
    private readonly WireMockServer _wireMockServer;
    private readonly HttpClient _httpClient;
    private readonly PaymentService _paymentService;

    public PaymentIntegrationTests()
    {
        _wireMockServer = WireMockServer.Start();
        _httpClient = new HttpClient { BaseAddress = new Uri(_wireMockServer.Urls[0]) };
        _paymentService = new PaymentService(_httpClient);
    }

    [Fact]
    public async Task ProcessPaymentAsync_SuccessfulResponse_ReturnsSuccess()
    {
        // Arrange
        _wireMockServer
            .Given(Request.Create()
                .WithPath("/api/payments")
                .WithMethod("POST")
                .WithBodyAsJson(new { amount = 100.00, cardToken = "token123" }))
            .RespondWith(Response.Create()
                .WithStatusCode(200)
                .WithBodyAsJson(new { transactionId = "12345", status = "approved" }));

        var request = new PaymentRequest { Amount = 100.00m, CardToken = "token123" };

        // Act
        var result = await _paymentService.ProcessPaymentAsync(request);

        // Assert
        Assert.True(result.IsSuccessful);
        Assert.Equal("12345", result.TransactionId);
    }

    [Fact]
    public async Task ProcessPaymentAsync_ServiceError_ReturnsFailure()
    {
        // Arrange
        _wireMockServer
            .Given(Request.Create()
                .WithPath("/api/payments")
                .WithMethod("POST"))
            .RespondWith(Response.Create()
                .WithStatusCode(500)
                .WithBodyAsJson(new { error = "Internal server error" }));

        var request = new PaymentRequest { Amount = 100.00m, CardToken = "token123" };

        // Act
        var result = await _paymentService.ProcessPaymentAsync(request);

        // Assert
        Assert.False(result.IsSuccessful);
        Assert.Contains("Internal server error", result.ErrorMessage);
    }

    public void Dispose()
    {
        _wireMockServer?.Stop();
        _httpClient?.Dispose();
    }
}
```

5.3.6 React Native Mocking Implementation

Jest with Mock Service Worker

Implement comprehensive API mocking for React Native applications:

```
// API service interface
export interface UserService {
  fetchUser(id: string): Promise<User>;
  updateUser(id: string, updates: Partial<User>): Promise<User>;
}

// Component under test
interface UserProfileProps {
  userId: string;
  userService: UserService;
}

export const UserProfile: React.FC<UserProfileProps> = ({ userId, userService }) => {
  const [user, setUser] = useState<User | null>(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState<string | null>(null);
```

```
  useEffect(() => {
    const loadUser = async () => {
      try {
        setLoading(true);
        const userData = await userService.fetchUser(userId);
        setUser(userData);
      } catch (err) {
        setError('Failed to load user data');
      } finally {
        setLoading(false);
      }
    };

    loadUser();
  }, [userId, userService]);

  if (loading) return <ActivityIndicator testID="loading-indicator" />;
  if (error) return <Text testID="error-message">{error}</Text>;
  if (!user) return <Text testID="no-user">User not found</Text>;

  return (
    <View testID="user-profile">
      <Text testID="user-name">{user.name}</Text>
      <Text testID="user-email">{user.email}</Text>
    </View>
  );
};

// Jest test implementation
import { render, waitFor } from '@testing-library/react-native';
import { UserProfile } from '../UserProfile';
import { UserService } from '../services/UserService';

describe('UserProfile Component', () => {
  const mockUserService: jest.Mocked<UserService> = {
    fetchUser: jest.fn(),
    updateUser: jest.fn(),
  };

  beforeEach(() => {
    jest.clearAllMocks();
  });

  it('displays user information when loaded successfully', async () => {
    // Arrange
    const mockUser = {
      id: '123',
      name: 'John Doe',
      email: 'john.doe@example.com'
    };

    mockUserService.fetchUser.mockResolvedValue(mockUser);

    // Act
    const { getByTestId } = render(
      <UserProfile userId="123" userService={mockUserService} />
    );

    // Assert
    await waitFor(() => {
      expect(getByTestId('user-name')).toHaveTextContent('John Doe');
      expect(getByTestId('user-email')).toHaveTextContent('john.doe@example.com');
    });

    expect(mockUserService.fetchUser).toHaveBeenCalledWith('123');
  });

  it('displays error message when service fails', async () => {
    // Arrange
    mockUserService.fetchUser.mockRejectedValue(new Error('Network error'));

    // Act
    const { getByTestId } = render(
      <UserProfile userId="123" userService={mockUserService} />
    );

    // Assert
    await waitFor(() => {
      expect(getByTestId('error-message')).toHaveTextContent('Failed to load user data');
    });
  });
});
```

MSW Integration for Network Mocking

Implement Mock Service Worker for comprehensive network request mocking:

```
// MSW handlers setup
import { rest } from 'msw';
import { setupServer } from 'msw/node';
```

```
export const handlers = [
  rest.get('/api/users/:id', (req, res, ctx) => {
    const { id } = req.params;

    return res(
      ctx.status(200),
      ctx.json({
        id,
        name: 'John Doe',
        email: 'john.doe@example.com',
        avatar: 'https://example.com/avatar.jpg'
      })
    );
  }),

  rest.put('/api/users/:id', async (req, res, ctx) => {
    const { id } = req.params;
    const updates = await req.json();

    return res(
      ctx.status(200),
      ctx.json({
        id,
        ...updates,
        updatedAt: new Date().toISOString()
      })
    );
  }),

  rest.get('/api/users/:id/orders', (req, res, ctx) => {
    const { id } = req.params;

    return res(
      ctx.status(200),
      ctx.json([
        {
          id: 'order1',
          userId: id,
          amount: 99.99,
          status: 'completed'
        }
      ])
    );
  })
];

export const server = setupServer(...handlers);

// Test setup configuration
import { server } from './mocks/server';

beforeAll(() => server.listen({ onUnhandledRequest: 'error' }));
afterEach(() => server.resetHandlers());
afterAll(() => server.close());

// Integration test with MSW
describe('User Service Integration', () => {
  it('fetches and displays user with orders', async () => {
    // Arrange
    const { getByTestId } = render(<UserProfileScreen userId="123" />);

    // Act & Assert
    await waitFor(() => {
      expect(getByTestId('user-name')).toHaveTextContent('John Doe');
      expect(getByTestId('order-count')).toHaveTextContent('1 order');
    });
  });

  it('handles server error gracefully', async () => {
    // Arrange
    server.use(
      rest.get('/api/users/123', (req, res, ctx) => {
        return res(ctx.status(500), ctx.json({ error: 'Internal server error' }));
      })
    );

    const { getByTestId } = render(<UserProfileScreen userId="123" />);

    // Act & Assert
    await waitFor(() => {
      expect(getByTestId('error-message')).toHaveTextContent('Failed to load user data');
    });
  });
});
```

React Native Component Testing

Implement comprehensive component testing with mock dependencies:

```
// Service hook with error handling
export const useUserData = (userId: string) => {
  const [user, setUser] = useState<User | null>(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState<string | null>(null);

  const fetchUser = useCallback(async () => {
    try {
      setLoading(true);
      setError(null);
      const userData = await userService.fetchUser(userId);
      setUser(userData);
    } catch (err) {
      setError(err instanceof Error ? err.message : 'Unknown error');
    } finally {
      setLoading(false);
    }
  }, [userId]);

  useEffect(() => {
    fetchUser();
  }, [fetchUser]);

  return { user, loading, error, refetch: fetchUser };
};

// Component test with custom hook mocking
jest.mock('../hooks/useUserData');

describe('UserProfileScreen', () => {
  const mockUseUserData = useUserData as jest.MockedFunction<typeof useUserData>;

  beforeEach(() => {
    jest.clearAllMocks();
  });

  it('renders user profile when data loads successfully', () => {
    // Arrange
    mockUseUserData.mockReturnValue({
      user: {
        id: '123',
        name: 'Jane Smith',
        email: 'jane.smith@example.com',
        avatar: 'https://example.com/avatar.jpg'
      },
      loading: false,
      error: null,
      refetch: jest.fn()
    });

    // Act
    const { getByTestId } = render(<UserProfileScreen userId="123" />);

    // Assert
    expect(getByTestId('user-name')).toHaveTextContent('Jane Smith');
    expect(getByTestId('user-email')).toHaveTextContent('jane.smith@example.com');
    expect(getByTestId('user-avatar')).toBeTruthy();
  });

  it('shows loading state while fetching data', () => {
    // Arrange
    mockUseUserData.mockReturnValue({
      user: null,
      loading: true,
      error: null,
      refetch: jest.fn()
    });

    // Act
    const { getByTestId } = render(<UserProfileScreen userId="123" />);

    // Assert
    expect(getByTestId('loading-indicator')).toBeTruthy();
  });

  it('displays error message and retry button when fetch fails', () => {
    // Arrange
    const mockRefetch = jest.fn();
    mockUseUserData.mockReturnValue({
      user: null,
      loading: false,
      error: 'Network connection failed',
      refetch: mockRefetch
    });

    // Act
    const { getByTestId } = render(<UserProfileScreen userId="123" />);

    // Assert
    expect(getByTestId('error-message')).toHaveTextContent('Network connection failed');

    // Test retry functionality
    fireEvent.press(getByTestId('retry-button'));
    expect(mockRefetch).toHaveBeenCalledTimes(1);
```

```
  });
});
```

5.3.7 Containerized Mock Services with Podman

Docker Mock Service Configuration

Create containerized mock services using Red Hat UBI images:

```
# Mock API service Dockerfile
FROM registry.access.redhat.com/ubi9/nodejs-18:latest

USER root

# Install WireMock standalone
RUN dnf update -y && \
    dnf install -y java-11-openjdk-headless wget && \
    dnf clean all

WORKDIR /app

# Download WireMock standalone JAR
RUN wget https://repo1.maven.org/maven2/com/github/tomakehurst/wiremock-jre8-standalone/2.35.0/wiremock-jre8-standalone-2.35.0.jar \
    -O wiremock-standalone.jar

# Copy mock configuration
COPY mappings/ /app/mappings/
COPY __files/ /app/__files/

USER 1001

EXPOSE 8080

CMD ["java", "-jar", "wiremock-standalone.jar", "--port", "8080", "--verbose"]
```

Mock Service Configuration Files

Configure WireMock mappings for containerized mock services:

```
// mappings/user-api.json
{
  "mappings": [
    {
      "request": {
        "method": "GET",
        "urlPathPattern": "/api/users/([0-9]+)"
      },
      "response": {
        "status": 200,
        "headers": {
          "Content-Type": "application/json"
        },
        "bodyFileName": "user-response.json",
        "transformers": ["response-template"]
      }
    },
    {
      "request": {
        "method": "POST",
        "urlPath": "/api/users",
        "bodyPatterns": [
          {
            "matchesJsonPath": "$.name"
          }
        ]
      },
      "response": {
        "status": 201,
        "headers": {
          "Content-Type": "application/json"
        },
        "body": "{\"id\": \"{{randomValue length=10 type='ALPHANUMERIC'}}\", \"name\": \"{{jsonPath request.body '$.name'}}\", \"createdAt\": \"{{now}}\"}",
        "transformers": ["response-template"]
      }
    }
  ]
}
```

```
// __files/user-response.json
{
  "id": "123",
  "name": "Mock User",
  "email": "mock.user@example.com",
  "avatar": "https://mock-service/avatars/default.jpg",
  "preferences": {
```

```
    "theme": "dark",
    "notifications": true
  },
  "metadata": {
    "lastLogin": "{{now format='yyyy-MM-dd HH:mm:ss'}}",
    "loginCount": "{{randomValue length=2 type='NUMERIC'}}"
  }
}
```

Podman Compose Configuration

Configure containerized mock services with Podman Compose:

```
# compose-mocks.yml
version: '3.8'

services:
  mock-user-api:
    build:
      context: ./mocks/user-api
      dockerfile: Dockerfile
    ports:
      - "8080:8080"
    environment:
      - WIREMOCK_OPTIONS=--global-response-templating
    volumes:
      - ./mocks/user-api/mappings:/app/mappings:ro
      - ./mocks/user-api/__files:/app/__files:ro
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8080/__admin/health"]
      interval: 30s
      timeout: 10s
      retries: 3
    networks:
      - mock-network

  mock-payment-api:
    build:
      context: ./mocks/payment-api
      dockerfile: Dockerfile
    ports:
      - "8081:8080"
    environment:
      - WIREMOCK_OPTIONS=--global-response-templating --async-response-enabled
    volumes:
      - ./mocks/payment-api/mappings:/app/mappings:ro
      - ./mocks/payment-api/__files:/app/__files:ro
    depends_on:
      - mock-user-api
    networks:
      - mock-network

networks:
  mock-network:
    driver: bridge
```

Integration Test Configuration

Configure integration tests with containerized mock services:

```
// Integration test with containerized mocks
public class ContainerizedMockTests : IAsyncLifetime
{
    private readonly IContainer _mockContainer;
    private readonly HttpClient _httpClient;

    public ContainerizedMockTests()
    {
        _mockContainer = new ContainerBuilder()
            .WithImage("mock-user-api:latest")
            .WithPortBinding(8080, true)
            .WithWaitStrategy(Wait.ForUnixContainer()
                .UntilHttpRequestIsSucceeded(r => r.ForPort(8080).ForPath("/__admin/health")))
            .Build();

        _httpClient = new HttpClient();
    }

    public async Task InitializeAsync()
    {
        await _mockContainer.StartAsync();
        var mockPort = _mockContainer.GetMappedPublicPort(8080);
        _httpClient.BaseAddress = new Uri($"http://localhost:{mockPort}");
    }

    [Fact]
    public async Task GetUser_ContainerizedMock_ReturnsExpectedData()
```

```
    {
        // Act
        var response = await _httpClient.GetAsync("/api/users/123");
        var content = await response.Content.ReadAsStringAsync();
        var user = JsonSerializer.Deserialize<User>(content);

        // Assert
        Assert.Equal(HttpStatusCode.OK, response.StatusCode);
        Assert.Equal("123", user.Id);
        Assert.Equal("Mock User", user.Name);
    }

    public async Task DisposeAsync()
    {
        _httpClient?.Dispose();
        await _mockContainer.DisposeAsync();
    }
}
```

Mock Service Management Scripts

Integrate mock services into development workflows:

```
#!/bin/bash
# scripts/start-mocks.sh

echo "Starting containerized mock services..."

# Build mock service images
podman build -t mock-user-api:latest ./mocks/user-api/
podman build -t mock-payment-api:latest ./mocks/payment-api/

# Start mock services with Podman Compose
podman-compose -f compose-mocks.yml up -d

# Wait for services to be ready
echo "Waiting for mock services to be ready..."
until curl -sf http://localhost:8080/__admin/health > /dev/null; do
    sleep 2
done

until curl -sf http://localhost:8081/__admin/health > /dev/null; do
    sleep 2
done

echo "Mock services are ready!"
echo "User API: http://localhost:8080"
echo "Payment API: http://localhost:8081"
echo "WireMock Admin: http://localhost:8080/__admin/"
```

```
#!/bin/bash
# scripts/stop-mocks.sh

echo "Stopping containerized mock services..."
podman-compose -f compose-mocks.yml down

echo "Cleaning up mock service images..."
podman rmi mock-user-api:latest mock-payment-api:latest 2>/dev/null || true

echo "Mock services stopped and cleaned up."
```

5.3.8 Best Practices and Recommendations

Mock Service Design Principles

Service Isolation: Design mock services to operate independently without external dependencies, ensuring reliable test execution in any environment.

Data Consistency: Maintain consistent mock data across test scenarios while supporting dynamic response generation for varied test conditions.

Performance Optimization: Configure mock services for rapid response times to maintain fast test execution cycles and immediate feedback.

Security Compliance: Implement mock services using security-hardened base images and follow container security best practices.

Development Process Integration

Continuous Integration: Integrate mock services into CI/CD pipelines using containerized deployment strategies that support parallel test execution.

Local Development: Provide simple scripts and documentation for developers to start and stop mock services during local development cycles.

environment Parity: Ensure mock service behavior closely matches production system behavior to prevent integration issues.

Version Management: Maintain mock service versions alongside application versions to support parallel development and testing workflows.

Testing Strategy Implementation

Test Pyramid Alignment: Use unit-level mocks for rapid feedback and integration-level mocks for comprehensive workflow validation.

Scenario Coverage: Design mock services to support comprehensive test scenario coverage including success paths, error conditions, and edge cases.

Contract Validation: Implement contract testing to ensure mock services accurately represent real system behavior and API specifications.

Monitoring and Observability: Include logging and monitoring capabilities in mock services to support test debugging and system understanding.

5.4 Performance Testing in Containerized Environments

5.4.1 What is Performance Testing?

Performance testing is a non-functional testing technique that measures the performance of an application under a specific workload. It helps to identify and eliminate performance bottlenecks in the software. The goal of performance testing is to ensure that the application is stable, scalable, and responsive under load.

Key metrics in performance testing include: * **Response Time:** The time it takes for the application to respond to a request. * **Throughput:** The number of requests the application can handle per unit of time. * **Error Rate:** The percentage of requests that result in an error. * **Resource Utilization:** CPU, memory, and network usage.

5.4.2 The Importance of Early Performance Testing (Shift-Left)

Traditionally, performance testing has been a final-stage activity, performed just before an application is released to production. This late-stage approach is risky and expensive. If performance issues are found at this stage, they can be difficult and costly to fix, potentially delaying the release.

Adopting a "shift-left" approach to performance testing means integrating it into the early stages of the development lifecycle. By testing early and often, developers can get rapid feedback on the performance implications of their code changes. This allows for a more iterative and proactive approach to performance optimization.

5.4.3 Local Performance Testing with Podman

Containers provide a lightweight and consistent environment for local performance testing. Developers can easily create a containerized version of their application and its dependencies, allowing them to run performance tests on their own machines using `podman`.

Using Podman for Local Testing

With `podman`, you can run your application and a load testing tool in separate containers on the same network, simulating a multi-service environment. This is ideal for testing how your application behaves under load.

Here's a simple workflow for local performance testing using `podman` and `k6`, a modern and powerful load testing tool:

1. **Create a network:**

```
podman network create my-test-network
```

2. **Build and run your application container:** Assuming you have a `Containerfile` for your application, build the image and run it on the network you created.

```
podman build -t my-app .
podman run -d --name my-app-container --network my-test-network -p 8080:8080 my-app
```

3. **Write a `k6` test script:** Create a file named `script.js` to define your load test.

```
import http from 'k6/http';
import { sleep } from 'k6';

export default function () {
  http.get('http://my-app-container:8080');
  sleep(1);
}
```

4. **Run the `k6` load tester container:** Run the official `k6` container, mounting your script and connecting it to the same network.

```
podman run -i --rm --network my-test-network -v $(pwd):/scripts grafana/k6 run /scripts/script.js
```

This setup uses `k6` to send requests to your application container (`my-app-container`) over the shared `podman` network. `k6` provides detailed output on response times, request rates, and potential errors, offering much more insight than older tools like `ab`.

5.4.4 Performance Testing on OpenShift

OpenShift provides a powerful platform for running and managing containerized applications. It also offers various tools and techniques for performance testing.

Using `hey` for Simple Load Testing

`hey` is a simple and effective command-line tool for load testing HTTP endpoints. It can be run from your local machine against an application running on OpenShift, or it can be run from within a pod in the OpenShift cluster.

To run `hey` from your local machine, you first need to expose your application's service as a route in OpenShift. Once you have the route, you can use `hey` to generate load:

```
# Install hey (on macOS)
brew install hey

# Run a load test
hey -n 2000 -c 50 -m GET https://my-app-route.openshift.com/
```

This command sends 2000 requests with a concurrency of 50 to the specified URL.

Robust Cloud-Native Tooling

For more advanced performance testing scenarios, you can leverage a variety of powerful, open-source, cloud-native tools:

- **k6:** A modern, developer-centric load testing tool designed for high-performance testing. Tests are written in JavaScript (or TypeScript), making them easy to write and maintain. `k6` provides excellent features for goal-oriented testing (e.g., setting thresholds for response times or error rates) and can be easily integrated into CI/CD pipelines. It can be run in a pod on OpenShift to generate load from within the cluster.
- **JMeter:** A long-standing and feature-rich performance testing tool with a large community. While traditionally a UI-driven tool, JMeter can be run in a non-GUI mode for automation and can be containerized for distributed load testing on Kubernetes/OpenShift.
- **Locust:** An easy-to-use, scriptable performance testing tool where you define user behavior in Python code. It's designed for testing websites and other systems and can simulate millions of simultaneous users. Its code-based approach makes it a favorite among developers.

5.4.5 Conclusion

Performance testing is a critical part of the software development lifecycle. By embracing a shift-left approach and leveraging containers and cloud-native tooling, teams can build more performant and resilient applications. Early and continuous performance testing helps to de-risk releases and ensures a better user experience.

6. 5. Red Hat Ecosystem

6.1 Red Hat Software Collections (RHSCL) and Modern Supply Chain Security

Red Hat Software Collections (RHSCL) was a Red Hat offering that provided developers with the latest stable versions of a wide range of development tools, dynamic languages, and open-source databases for Red Hat Enterprise Linux (RHEL).

6.1.1 What was Red Hat Software Collections?

RHSCL allowed developers to work with newer software versions than those included in the base RHEL system. A key feature was the ability to install and use multiple versions of the same software concurrently without affecting the system's default packages. This was achieved by installing the collection's packages in the `/opt/` directory, preventing conflicts with the base system tools.

Benefits of RHSCL included: * **Access to recent software versions:** Developers could use the latest features in languages like Python, PHP, Ruby, and Node.js, as well as databases such as MongoDB, MariaDB, and PostgreSQL. * **Stability and support:** RHSCL provided a stable and supported environment with a support lifecycle of two to three years for each collection. * **Container-friendly:** Many collections were available as container images.

6.1.2 How did RHSCL help with Supply Chain Security?

RHSCL provided a degree of supply chain security by:

- **Providing a trusted source:** The software collections were provided and signed by Red Hat, ensuring they came from a trusted source.
- **Security Advisories:** Red Hat issued Critical and Important Security Errata Advisories (RHSAs) and Urgent Priority Bug Fix Errata Advisories (RHBAs) for the collections.
- **Defined Lifecycle:** Each collection had a defined support lifecycle, so users knew for how long security updates would be provided.

6.1.3 The Evolution to Application Streams and Red Hat Trusted Software Supply Chain

Starting with RHEL 8, the content traditionally delivered via Software Collections is now part of **Application Streams**.

For a more comprehensive and modern approach to software supply chain security, Red Hat has introduced the **Red Hat Trusted Software Supply Chain**. This solution integrates security into every phase of the software development lifecycle (SDLC).

Components of the Red Hat Trusted Software Supply Chain:

- **Red Hat Trusted Content:** A curated catalog of trusted and signed open-source software packages.
- **Red Hat Trusted Application Pipeline:** A security-focused CI/CD service for containerized applications.
- **Red Hat Trusted Artifact Signer:** Allows developers to cryptographically sign and verify software artifacts.
- **Red Hat Trusted Profile Analyzer:** A tool for managing and analyzing Software Bill of Materials (SBOMs) and Vulnerability Exploitability Exchange (VEX) documents.
- **Red Hat Advanced Cluster Security for Kubernetes (ACS):** Provides Kubernetes-native security features.
- **Red Hat Quay:** A security-focused, scalable private container registry.

6.1.4 How to Get Started

Using Red Hat Software Collections (RHEL 7 and earlier)

To use a specific collection, you can use the `scl` utility to enable it. This creates a new shell environment with the necessary environment variables updated.

```
# To enable a software collection
scl enable <collection_name> bash
```

Engaging with the Red Hat Trusted Software Supply Chain

The Red Hat Trusted Software Supply Chain is a comprehensive solution that is layered onto application platforms like Red Hat OpenShift. To get started, you would typically:

1. **Use Red Hat OpenShift:** as your Kubernetes platform.
2. **Integrate Red Hat Trusted Content:** into your development process.
3. **Implement Red Hat Trusted Application Pipeline:** for secure CI/CD.
4. **Utilize Red Hat Advanced Cluster Security (ACS):** to secure your Kubernetes environment.