A developer's guide to moving from monoliths to microservices

Table of contents

Introduction

Cloud-native development is increasingly important for organizations that want to exploit microservices through containerized and orchestrated cloud deployments. However, completely rewriting all legacy applications is seldom feasible due to time and cost requirements. Red Hat's stepwise cloud migration approach reuses existing functionality and data as much as possible. The process moves existing workloads to a modern deployment platform and ultimately applies new processes, products, and technologies to modernize the application.

Modernizing monolithic applications starts with a "lift-and-shift" process as a first step to cloud-native development. Described in A developer's guide to lift-and-shift cloud migration, this process includes:

- Containerizing existing monolithic workloads.
- Deploying the workload on Red Hat® OpenShift® Container Platform, Red Hat's Kubernetes application platform solution.
- Retaining external integrations and data on the legacy platform.

Once those steps are accomplished, developers can begin to "strangle the monolith." As shown in Figure 1, this process involves incrementally replacing in-app functionality with microservices that are lighter, faster, and easier to maintain. As functionality is replaced, portions of the monolith can be optionally removed or retired. Developers can also introduce new functionality through new microservices during this process to make the application more attractive to customers or business stakeholders.

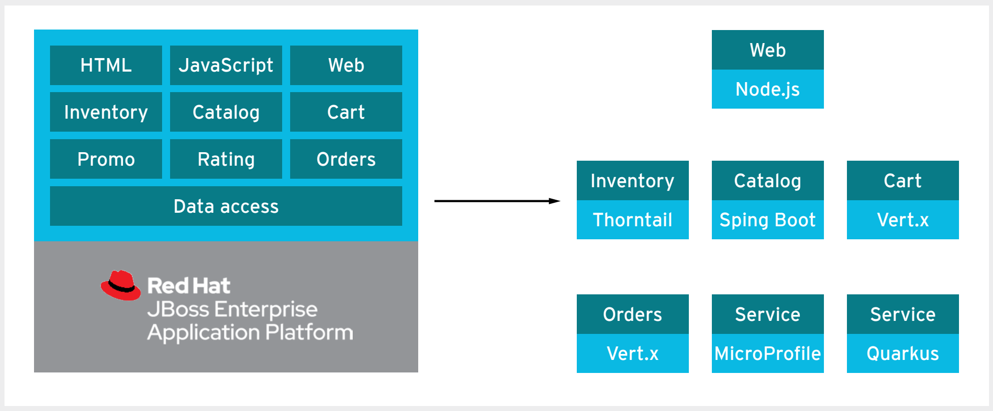


Figure 1. Application modernization involves selectively replacing in-app functionality with microservices.

Identifying functionality for refactoring as a microservice

Moving from monoliths to microservices presents development teams with the opportunity to refactor their code—either improving existing functionality or introducing new functionality while reducing the inherent complexity of the monolith. Unfortunately, the process of deciding what functionality to extract into a microservice can be complex. While viewing software as a single unified model would be ideal, it is often not practical with large models and teams.

The proven domain-driven design concept offers an effective strategy that connects software implementation to an evolving model, potentially based on microservices. In particular, using the bounded context pattern can be a constructive way to divide application functionality into meaningful component parts that are candidates for refactoring as microservices. Through this approach, large, complex models are divided into bounded contexts, with specific interrelationships between contexts.

This guide provides discussion around the process of "strangling the monolith"—extending and augmenting a monolith that implements a fictitious "Coolstore" online store with additional microservices. Once it has been lifted and shifted, the monolith continues to function within Red Hat OpenShift Container Platform, while new microservices are seamlessly introduced to replace or augment in-app functionality (Figure 2). In this example, new services are added to the monolith based on open source runtimes including:

1. An inventory microservice built with Thorntail, Red Hat's implementation of Eclipse MicroProfile.
2. A catalog microservice built with Spring Boot.
3. A shopping cart microservice and database built with Eclipse Vert.x.

Future services could then be added as desired to replace or extend application functionality.

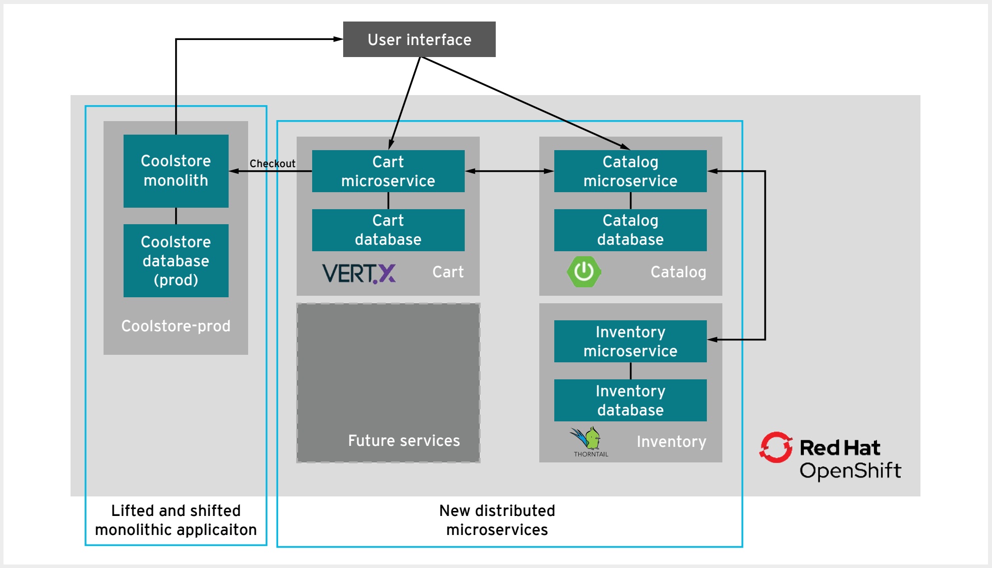


Figure 2. Modernizing a lifted-and-shifted monolith with microservices, all running on Red Hat OpenShift Container Platform.

Red Hat Runtimes

Red Hat Runtimes provides developers with multiple modernization options to enable a smooth transition to the cloud for existing applications. Teams benefit from using a single cloud-native development platform for creating new applications and transitioning from monoliths to microservices at their own pace. Multiple runtimes and frameworks are supported in Red Hat Runtimes.

- Runtimes include Red Hat JBoss Enterprise Application Platform, Eclipse Vert.x, Thorntail, and Node.js.
- Frameworks include Spring Boot, Netflix Ribbon, and Netflix Hystrix.

The product's launch service helps developers get up and running quickly in the cloud through a number of ready-to-run examples—or mission boosters—that demonstrate the power of Red Hat Runtimes (Figure 3).

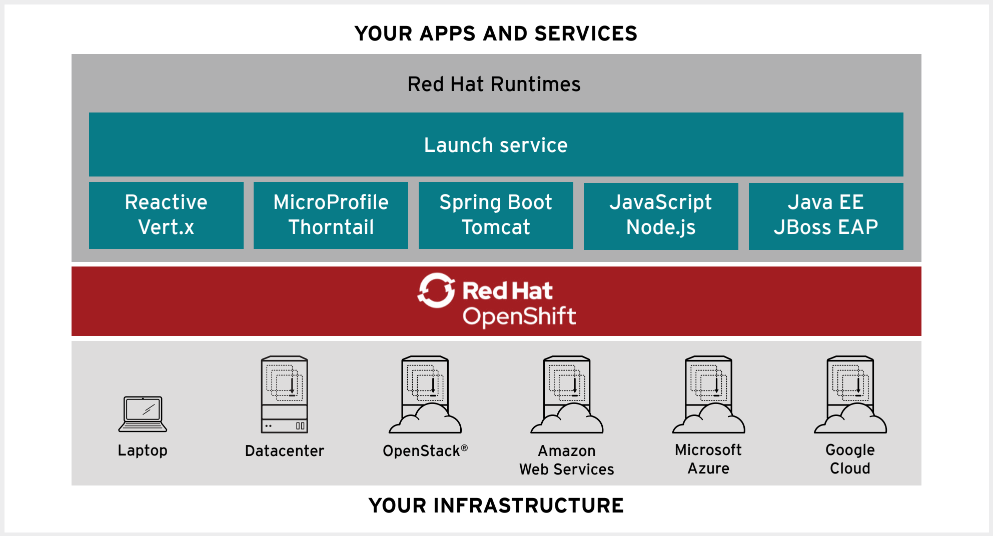


Figure 3. Red Hat Runtimes simplifies cloud-native development.

Choosing open source technology for microservices

The process documented in A developer's guide to lift-and-shift cloud migration describes how an existing monolithic Java™ EE application could be moved to the cloud with Red Hat JBoss Enterprise Application Platform and Red Hat OpenShift Container Platform, demonstrating some of the power of Red Hat OpenShift for existing applications. Moving to microservices, in turn, involves breaking down the application into smaller pieces for greater parallelization and autonomy. The process can reduce time to value, with the eventual goal of redesigning the entire application as a set of distributed microservices. Different technologies may be ideal for implementing different microservices. For example:

- Thorntail, which implements the MicroProfile specification, is ideal for refactoring existing Java EE applications with microservices, as well as those using Red Hat JBoss Enterprise Application Platform.
- Spring Boot can help build web applications that use the Spring ecosystem, resulting in a short path to production.
- Elipse Vert.x facilitates building reactive microservices for mobile or web apps, application programming interfaces (APIs), Internet of Things (IoT) apps, and real-time apps.
- Node.js provides server-side JavaScript ideal for building mobile and web apps, as well as building APIs and real-time apps.

Thorntail (Eclipse Microprofile)

Thorntail is a great place to start for existing Java EE applications. Existing skills as a Java EE developer likewise translate naturally to the world of Thorntail. As a MicroProfile implementation,[1] Thorntail provides a lightweight implementation with specialized tooling for developers. Thorntail "pieces" support the compositional aspect in Thorntail and provide a specific piece of functionality embodied as an Apache Maven artifact, offering:

- A way to add API dependencies (e.g., JAX-RS).
- The ability to configure the system with reasonable defaults.
- Discovery of other components as a part of a topology.
- A means to alter deployments.

Pieces can be auto-detected or explicitly declared.

Thorntail provides cloud-native support, including:

- Health checks.
- Externalized configuration.
- Client-side discovery and load balancing.
- Circuit breaking and bulkheading.
- Logging, monitoring, tracing, and metrics.
- Enterprise-level security deployment with Keycloak.

In a typical scenario, a developer would implement one component of the monolith as a Thorntail microservice and deploy it to Red Hat OpenShift Container Platform. The new microservice would be deployed alongside the existing monolith, running in tandem with it. For example, Thorntail could be used to implement an inventory microservice and database (Figure 4).

1 *Eclipse MicroProfile is designed to optimize enterprise Java for the microservices architecture. https://microprofile.io.*

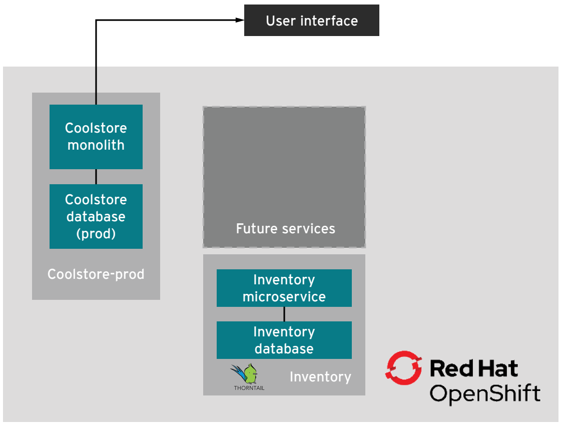


Figure 4. A new inventory microservice and database based on Thorntail runs alongside the existing monolithic application.

Building the service involves writing some code to create a domain model, service interface, and a RESTful endpoint to access inventory (Figure 5). When complete, the REST services define two endpoints:

- */services/inventory* that is accessible via HTTP GET, which will return all known product Inventory entities as JSON.
- */services/inventory/***<id>** that is accessible via HTTP GET, with the last path parameter being the product ID for which we want to check inventory status (e.g., */services/inventory/329299*).

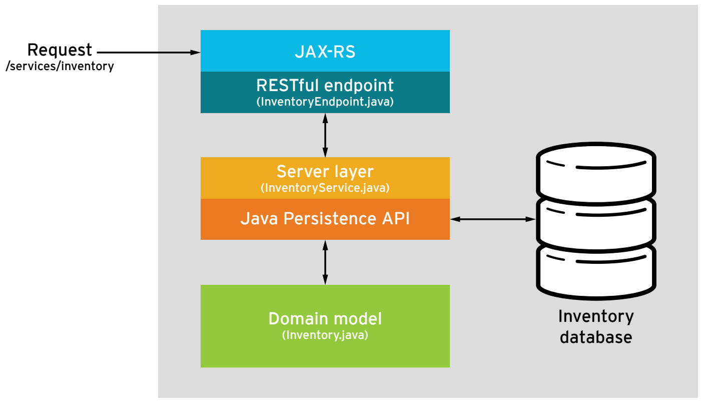


Figure 5. Inventory database and components.

The Spring Framework and Spring Boot

The previous scenario resulted in the creation of an inventory service and database, but so far we have not started truly *strangling* the monolith. The inventory service is never called directly by the user interface (UI)—since it is a back-end service that is used only by other back-end services. This section describes adding a catalog service that will, in turn, call the inventory service. Eventually, UI calls will be rerouted to the new service.

This example uses the Spring Framework, also available as a part of Red Hat Runtimes. Spring is one of the most popular Java frameworks and offers an alternative to the Java EE programming model. Reasons for choosing Spring or another platform mostly depend on personal preferences or existing knowledge. At their core, Spring and Java EE are similar.

Spring is popular for building applications based on microservices architectures. Spring Boot is a popular tool in the Spring ecosystem that helps with organizing and using third-party libraries together with Spring and also provides a mechanism for bootstrapping embeddable runtimes, like Apache Tomcat. Bootable applications (sometimes also called *fat jars*) fit the container model well. In Red Hat OpenShift Container Platform, responsibilities like starting, stopping, and monitoring applications are handled by the container platform instead of an application server. Like Thorntail, Spring provides full cloud-native functionality.

As a part of Red Hat Runtimes, Spring support includes:

- Red Hat testing and verification of Sprint Boot, Spring Cloud Kubernetes, Ribbon, and Hystrix.
- Full support for Tomcat, Hibernate, Apache CXF, single sign-on (with Keycloak), and messaging with Red Hat AMQ.
- Native Kubernetes and OpenShift integration (Spring Cloud) for service discovery via Kubernetes and Ribbon and Spring Cloud Config via ConfigMap.
- Developer tooling with launch.openshift.io and starters.

As an example, Spring Boot could be used to add a catalog microservice and database to our Coolstore online store application. The catalog microservice would, in turn, use the inventory microservice developed in the previous step (Figure 6).

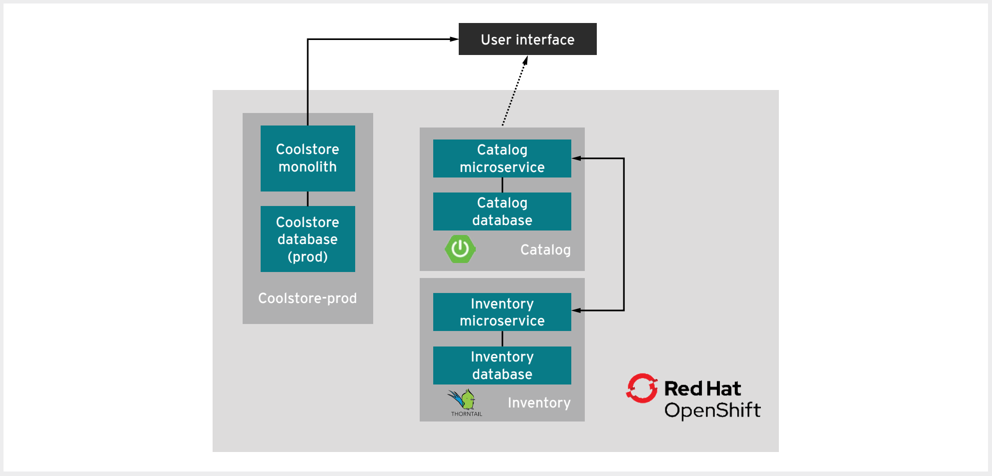


Figure 6. The catalog microservice built with Spring Boot communicates with the inventory service built with Thorntail.

Eclipse Vert.x

Reactive microservices provide responses to stimuli. Eclipse Vert.x is a toolkit used to build distributed and reactive systems. With an asynchronous nonblocking development model and a fundamental understanding of clustering in its core architecture, Vert.x offers:

- Simplified concurrency (event loop).
- Reactive microservices, web applications, and IoT.
- High-volume, low-latency applications.

Vert.x is event-driven and nonblocking, which means that applications in Vert.x can handle many concurrent requests using a small number of kernel threads. It supports everything from network utilities, sophisticated modern web applications, HTTP/REST microservices, and high-volume event processing to a full-blown back-end message-bus application. Vert.x is:

- Trusted. It is used by many different companies with apps that range from real-time gaming to banking.
- Lightweight. Vert.x core is around 650KB in size.
- Fast, as attested by independent performance results.
- Modular. When you need more functionality, just add the components you need and nothing more.

With Vert.x, you can:

- Scale your apps with minimal hardware.
- Create powerful apps the way you want in the language you want. Vert.x is not a restrictive framework or container, and it does not dictate how to write an application.

- Run your apps wherever you want. Vert.x is not an application server, and there is no monolithic Vert.x instance into which you deploy applications.
- Create lightweight, high-performance microservices.

In our example application, Vert.x could be used to implement a shopping cart microservice. As shown in Figure 7, this reactive microservice would:

- Communicate with the UI for user interaction.
- Communicate with the catalog microservice to aid user shopping.
- Communicate with the original Coolstore monolith for checkout.

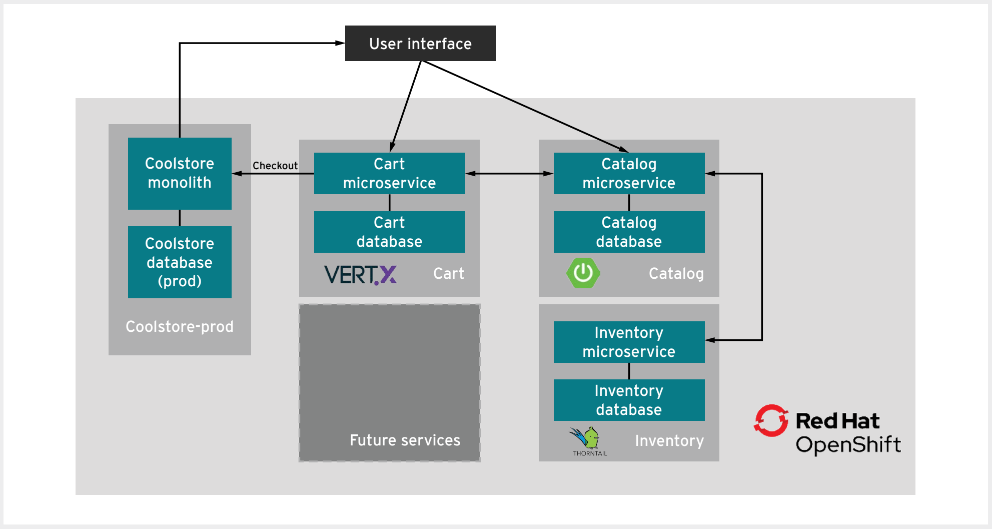
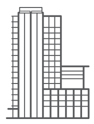


Figure 7. An adaptive shopping cart microservice built with Eclipse Vert.x communicates with the UI and the catalog microservice.

Summary

Red Hat Runtimes offers developers cloud-native development tools and platforms to help organizations move to microservices. After a monolith has been lifted and shifted to the cloud using Red Hat JBoss Enterprise Application Platform and Red Hat OpenShift Container Platform, microservices can be added to replace or augment aspects of the monolith, all while the monolith continues to operate. With platforms like Thorntail, which implements the MicroProfile specification, Spring Boot, Node.js, and Eclipse Vert.x, developers have control and choice for implementing microservices in the cloud.



NORTH AMERICA	EUROPE, MIDDLE EAST, AND AFRICA	ASIA PACIFIC	LATIN AMERICA
1 888 REDHAT1	00800 7334 2835	+65 6490 4200	+54 11 4329 7300
	europe@redhat.com	apac@redhat.com	info-latam@redhat.com