



Red Hat Reference Architecture Series

Microservice Architecture

Building microservices with JBoss EAP 7

Babak Mozaffari

Version 1.0, June 2016

Table of Contents

Comments and Feedback	2
Staying In Touch.....	2
Like us on Facebook:	2
Follow us on Twitter:	2
Plus us on Google+.....	2
1. Executive Summary.....	3
2. Microservice Architecture	4
2.1. Definition	4
2.2. Tradeoffs.....	5
2.2.1. Advantages	5
2.2.2. Disadvantages	5
2.3. Distributed Modularity Model.....	6
2.3.1. Overview.....	6
2.3.2. Monolithic Applications	7
2.3.3. Tactical Microservices	9
2.3.4. Strategic Microservices	11
2.3.5. Business-Driven Microservices	14
2.4. Cross-cutting concerns	16
2.4.1. Overview.....	16
2.4.2. Containerization	16
2.4.3. Service Discovery	16
2.4.4. Load Balancer	17
2.4.5. Cache	17
2.4.6. Throttling, Circuit Breaker, Composable Asynchronous Execution.....	17
2.4.7. Security	18
2.4.8. Monitoring and Management	18
2.4.9. Resilience Testing	18
2.5. Anatomy of a Microservice	19
3. Reference Architecture Environment.....	20
4. Creating the Environment	21
4.1. Prerequisites	21
4.2. Downloads	21
4.3. Installation	22
4.4. Configuration	22
4.4.1. Red Hat JBoss Core Services Apache HTTP Server	24
4.4.2. MySQL / MariaDB Database	28

4.4.3. JBoss Enterprise Application Platform	29
4.5. Deployment	30
4.6. Execution	31
5. Design and Development	38
5.1. Overview	38
5.2. Integrated Development Environment	38
5.2.1. JBoss Developer Studio	38
5.2.2. Creating a Maven Project	39
5.2.3. Configuring Java 8	41
5.3. Java Persistence API (JPA)	42
5.3.1. Overview	42
5.3.2. Persistence Unit	42
5.3.3. Persistence Entity	44
5.3.4. Database setup	47
5.4. RESTful API	53
5.4.1. Enabling JAX-RS support	53
5.4.2. RESTful Service	55
5.4.3. Transactional Behavior	58
5.4.4. Logging	60
5.4.5. Error handling	60
5.4.6. Resource API design	68
5.4.7. Other RESTful operations	77
5.4.8. Pessimistic Locking	80
5.4.9. Sales service	82
5.4.10. Sub-resources, RESTful relationships	95
5.4.11. Billing Service	103
5.4.12. Asynchronous Processing	105
5.4.13. Managed Executor Service	107
5.5. Aggregation/Presentation Layer	108
6. Conclusion	148
Appendix A: Revision History	149
Appendix B: Contributors	150

100 East Davie Street
Raleigh NC 27601 USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701
PO Box 13588
Research Triangle Park NC 27709 USA

Linux is a registered trademark of Linus Torvalds. Red Hat, Red Hat Enterprise Linux and the Red Hat "Shadowman" logo are registered trademarks of Red Hat, Inc. in the United States and other countries. Microsoft and Windows are U.S. registered trademarks of Microsoft Corporation. UNIX is a registered trademark of The Open Group. Intel, the Intel logo and Xeon are registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. OpenStack is the trademark of the OpenStack Foundation. All other trademarks referenced herein are the property of their respective owners.

© 2015 by Red Hat, Inc. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, V1.0 or later (the latest version is presently available at <http://www.opencontent.org/openpub/>).

The information contained herein is subject to change without notice. Red Hat, Inc. shall not be liable for technical or editorial errors or omissions contained herein.

Distribution of modified versions of this document is prohibited without the explicit permission of Red Hat Inc.

Distribution of this work or derivative of this work in any standard (paper) book form for commercial purposes is prohibited unless prior permission is obtained from Red Hat Inc.

The GPG fingerprint of the security@redhat.com key is: CA 20 86 86 2B D6 9D FC 65 F6 EC C4 21 91 80 CD DB 42 A6 0E

Send feedback to refarch-feedback@redhat.com

Comments and Feedback

In the spirit of open source, we invite anyone to provide feedback and comments on any of the reference architectures. Although we review our papers internally, sometimes issues or typographical errors are encountered. Feedback allows us to not only improve the quality of the papers we produce, but allows the reader to provide their thoughts on potential improvements and topic expansion to the papers. Feedback on the papers can be provided by emailing refarch-feedback@redhat.com. Please refer to the title within the email.

Staying In Touch

Join us on some of the popular social media sites where we keep our audience informed on new reference architectures as well as offer related information on things we find interesting.

Like us on Facebook:

<https://www.facebook.com/rhrefarch>

Follow us on Twitter:

<https://twitter.com/RedHatRefArch>

Plus us on Google+

<https://plus.google.com/114152126783830728030/>

1. Executive Summary

The *Information Technology* landscape is constantly shifting and evolving. Advancement in computer hardware has continually increased processing power and storage capacity, while network and internet connectivity has become faster and more widespread. Along with the proliferation of mobile devices, these factors have resulted in a global user-base for a large number of software services and applications.

Software designers and developers are not isolated from these changes; they must account for the experiences of the past as well as the characteristics of this ever-changing landscape to continually innovate in the way software is designed and delivered. The *microservices* architectural style is one such effort, aiming to apply some of the best practices learned in the past towards the requirements and the dynamically scalable deployment environments of certain software and services of the present and near-future.

Cloud computing has made tremendous progress in terms of cost and availability in recent years. The agile movement has brought the principles and practices of continuous integration and delivery to various organizations, and given rise to *DevOps*. Cultural changes continue to demand greater agility from IT organizations in responding to competitive threats, market opportunities, regulatory requirements, and security vulnerabilities. These multidimensional factors have paved the way for microservices and provided a sense of momentum for its adoption.

This reference architecture recites the basic tenets of a microservice architecture and analyzes some of the advantages and disadvantages of this approach. This paper expressly discourages a *one size fits all* mentality, instead envisioning various levels of modularity for services and deployment units.

The sample application provided with this reference architecture demonstrates Business-Driven Microservices. The design and development of this system is reviewed at length and the steps to create the environment are documented.

2. Microservice Architecture

2.1. Definition

Microservice Architecture (MSA) is a software architectural style that combines a mixture of well-established and modern patterns and technologies to achieve a number of desirable goals.

Some aspects, for example a *divide and conquer* strategy to decrease system complexity by increasing modularity, are universally accepted and have long been cornerstones of other competing paradigms.

Other choices carry trade-offs that have to be justified based on the system requirements as well as the overall system design.

General characteristics of microservices include:

- Applications are developed as a suite of small services, each running as an independent process in [its own logical machine](#) (or Linux container)
- Services are [built around capabilities](#): single responsibility principle
- Each microservice has a separate codebase and is owned by a separate team
- One can independently replace / upgrade / scale / deploy services
- Standard lightweight communication is used, often REST calls over HTTP
- Potentially heterogeneous environments are supported

2.2. Tradeoffs

The defining characteristic of a Microservice Architecture environment is that modular services are deployed individually and each can be replaced independent of other services or other instances of the same service. Modularity and other best practices yield a number of advantages but the most unique tradeoffs from MSA are the result of this characteristic.

2.2.1. Advantages

- Faster and simpler deployment and rollback with smaller services. Taking advantage of the divide and conquer paradigm in software delivery and maintenance.
- Ability to horizontally scale out individual services. Not sharing the same deployment platform with other services allows each service to be scaled out as needed.
- Selecting the right tool, language and technology per service, without having to conform to a homogeneous environment being dictated by shared infrastructure.
- Potential for fault isolation at microservice level by shielding services from common infrastructure failure due to the fault of one service. Where a system is designed to withstand the failure of some microservices, the result is Higher Availability for the system.
- Goes hand in hand with Continuous Delivery and Integration.
- Promotes DevOps culture with higher service self-containment and less common infrastructure maintenance.
- More autonomous teams lead to faster/better development.
- Facilitates A/B testing and canary deployment of services.
- Traditional divide and conquer benefits.

2.2.2. Disadvantages

The downsides of MSA are direct results of higher service distribution. There is also a higher cost to having less common infrastructure. Disadvantage may be enumerated as follows:

- Network reliability is always a concern.
- Less tooling / IDE support given the distributed nature.
- Tracing, monitoring and addressing cascading failures are complex.
- QA, particularly integration testing can be difficult.
- Debugging is always more difficult for distributed systems.
- Higher complexity – higher fixed cost and overhead.
- Heterogenous environments are difficult and costly to maintain.

2.3. Distributed Modularity Model

2.3.1. Overview

While modular design is a common best practice that is appropriate in just about all circumstances and environments, the logical and physical distribution of the modular units greatly vary, depending on the system architecture.

Some factors to consider:

- The number of developers: The [ideal size of a development team](#) is between 5 and 10 people and each team can focus on one or more microservices. In an organization with only 1 or 2 development teams, the case for decoupling the work is less compelling and the resulting overhead from the architectural choices may be too costly.
- Are you comfortable on the cutting edge of technology? In its specific shape and form, Microservice Architecture is a new paradigm with only a handful of success stories behind it. The tools and infrastructure to support MSA are neither abundant nor mature, and the cost of adoption is still high.
- Can you adapt your staffing to DevOps? One of the benefits of MSA is its amenability to a DevOps method and the resulting higher agility. This requires lines to be blurred between development and operations. Not every organization is prepared for the required cultural change.
- Do you have a production-grade cloud infrastructure: self-service, on-demand, elastic, API-based consumption model with multi-tenant billing capabilities? How easily can independent teams deploy services to production?
- How skilled are you at troubleshooting system errors? Like any distributed system, an MSA environment can be very difficult to analyze and troubleshoot.
- Can you afford [higher up-front costs](#)? Just about every software methodology and paradigm seeks to maximize the return on investment and minimize the costs. However, costs are not always evenly distributed in various stages of the software lifecycle. Individual service deployment and a distributed architecture increases complexity and the fixed cost associated with the environment.
- Do you have a network that can support the architecture? The distributed nature of an MSA environment puts more stress on the network and conversely, a more reliable network is required to support such an architecture.

2.3.2. Monolithic Applications

While many Microservices advocates may use the term *monolithic* disparagingly, this paper reserves judgement on this design and views it as the result of a series of legitimate trade-offs. This style of architecture may be preferable for certain situations and not for others.

Monolithic applications may be just as modular as microservices, but those modules are typically bundled as a single EAR or WAR file and deployed on a single application server and therefore the same logical machine. In this model, all the modules take advantage of the same infrastructure and maximize efficiency by minimize network traffic and latency. In some situations, it may even be possible to pass arguments by reference and avoid serialization and data transfer costs.

This diagram shows a traditional Java EE application deployed on a logical machine. It is emphasized that this single application consists of two web applications as well as three business services, each being modular (containing six embedded modules each):

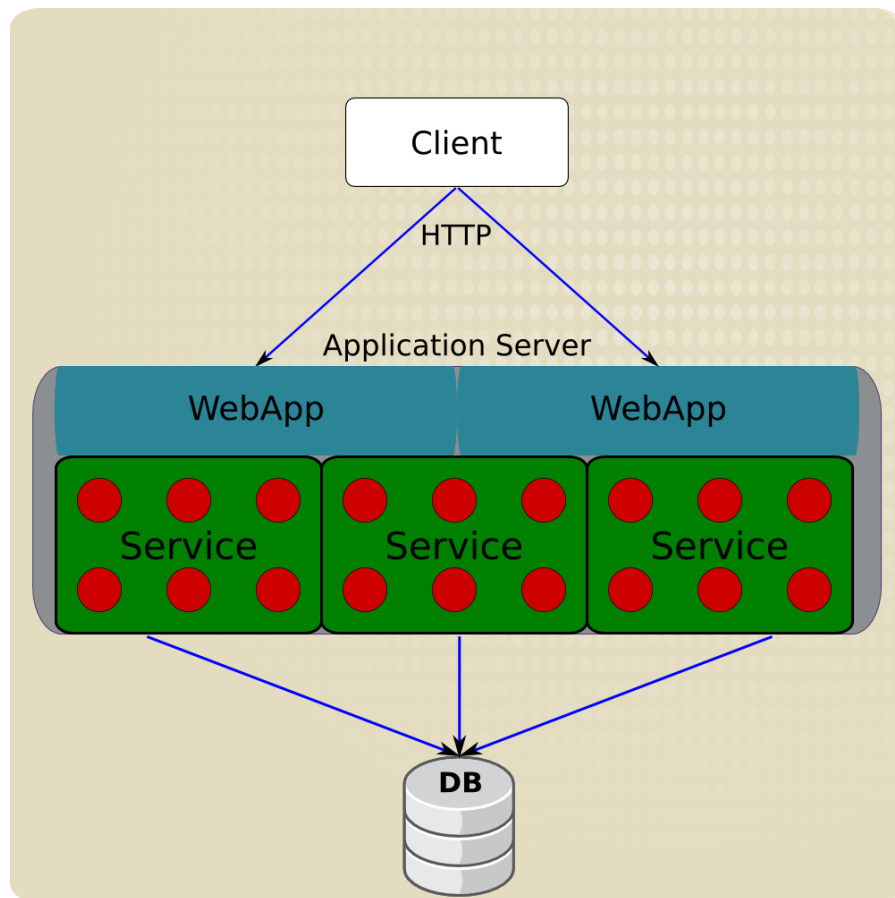


Figure 1. Java Enterprise Application

This deployment model minimizes overhead by sharing the application server and environment resources between various components.

Horizontal scaling of such an architecture is simple and often benefits from the clustering capabilities of the underlying application server. Most often, the entire environment is duplicated and the application server replicates any stateful application data that is held in memory:

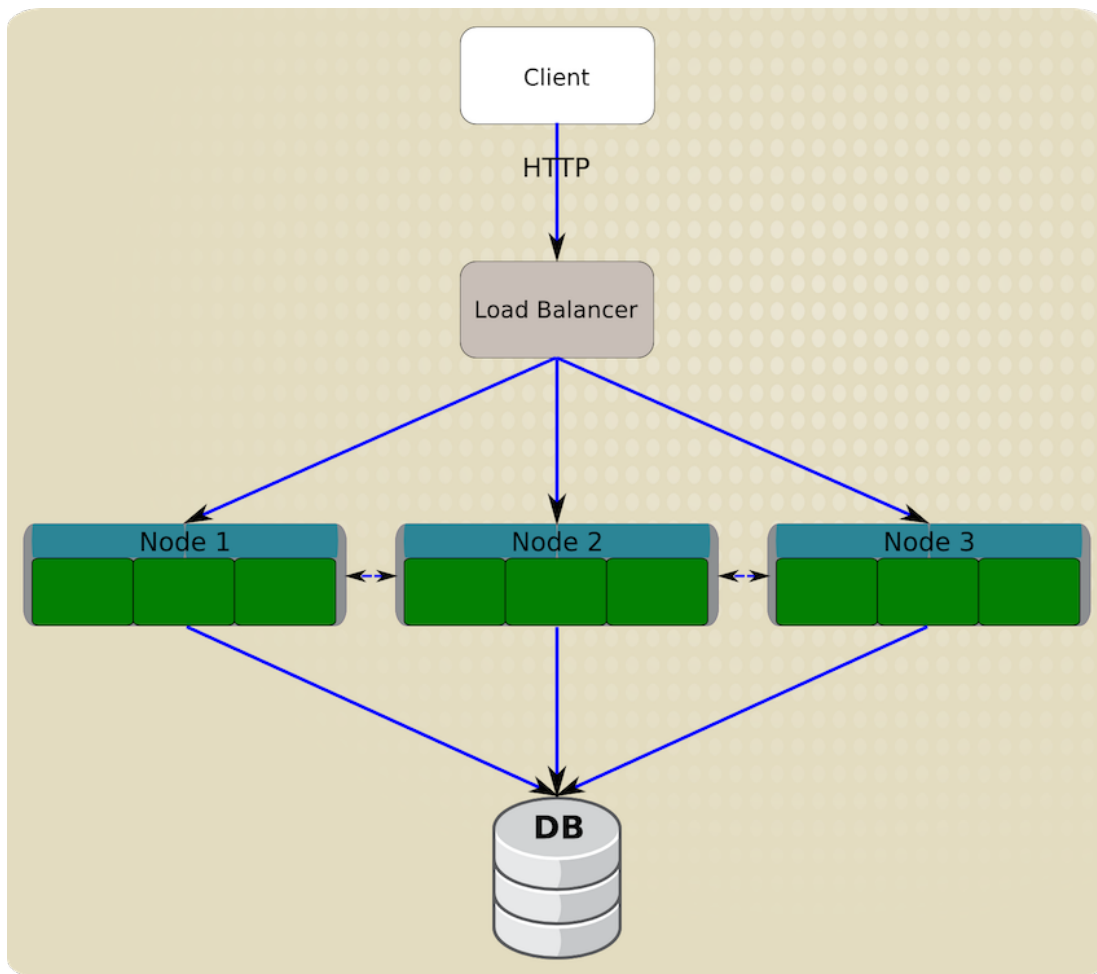


Figure 2. Clustered Java EE Application

The uniformity and consistency of the replicated environment can be as much a handicap, as it is an asset. Deployment, testing and maintenance is simplified and the consistency avoids various technical and logistics issues.

Things begin to change when one service is much less stable and requires more resources than others. Imagine that the first of the three business services is ten times more likely to hang or otherwise display unpredictable behavior. If that service can crash and bring down the server, it would also take down the other two services. Scaling out this service would likewise require scaling out the entire environment including services that may not have as much load or resource requirements. These issues are some of the biggest drivers of the microservice architecture.

2.3.3. Tactical Microservices

One possible strategy is to address the weaknesses of a traditional monolithic application architecture while continuing to take advantage of its benefits. Instead of proactively decomposing the application into microservices to allow separate lifecycle and deployment, isolate them or separately scale each out, some organizations prefer to take advantage of the common infrastructure and environment uniformity where possible, while explicitly identifying and extracting components that warrant separation. For example, if one of the business services in the application depicted in [Java Enterprise Application](#) is unstable or requires more resources, or if it is best maintained and upgraded as a small and separate unit that is managed by a dedicated team, it may be deployed separately. Similarly, a component within another business service may be extracted and separated:

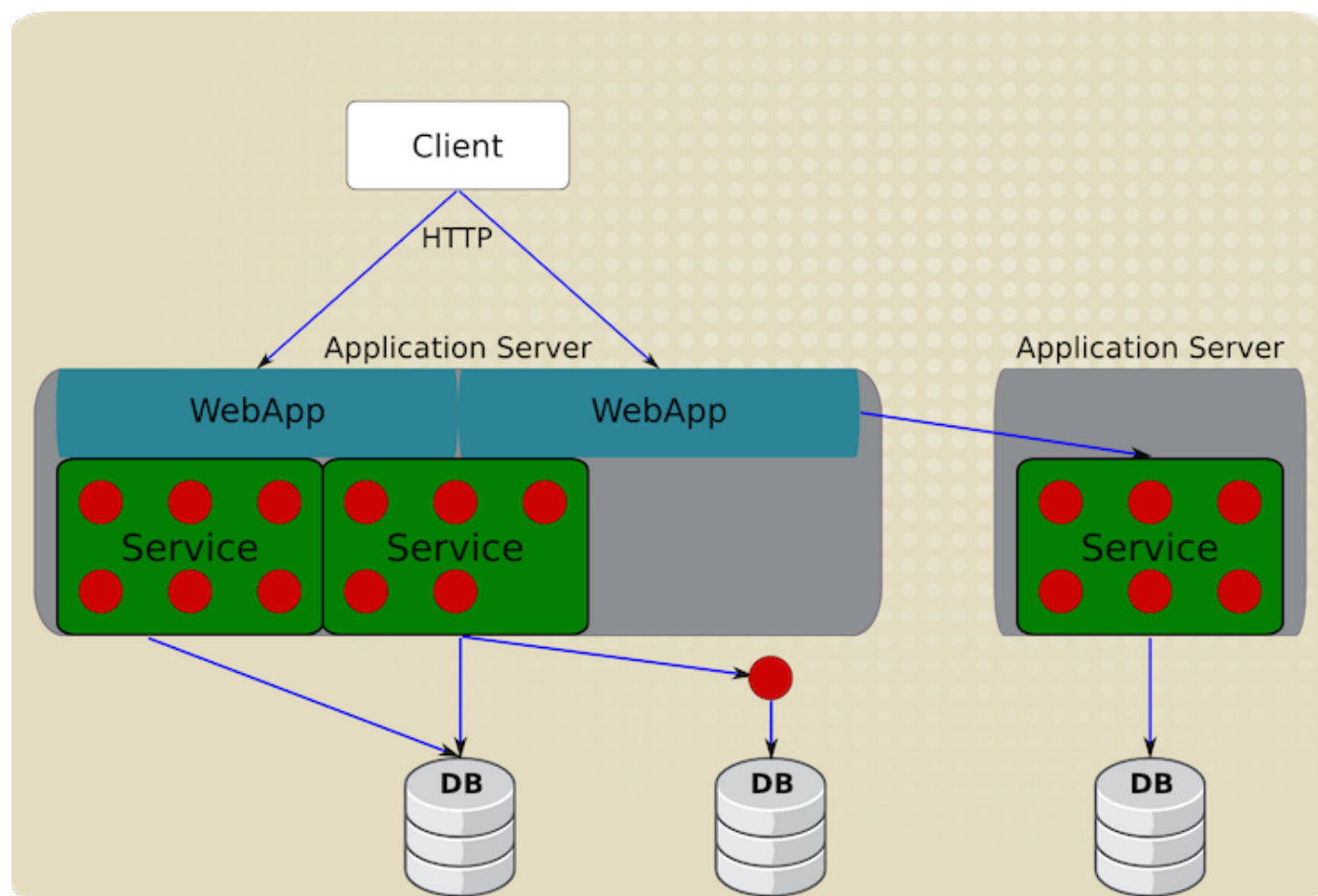


Figure 3. Tactical Microservices

Notice that in this architecture, each new deployment is self-encapsulated and includes its own persistence. The business service continues to be called from the web application, although new restrictions are imposed and this call is now necessarily a remote call. It is preferable to follow RESTful practices and communicate using XML or JSON, over HTTP or a similar transport.

This architecture allows the business service and the newly independent microservice to be scaled out separately:

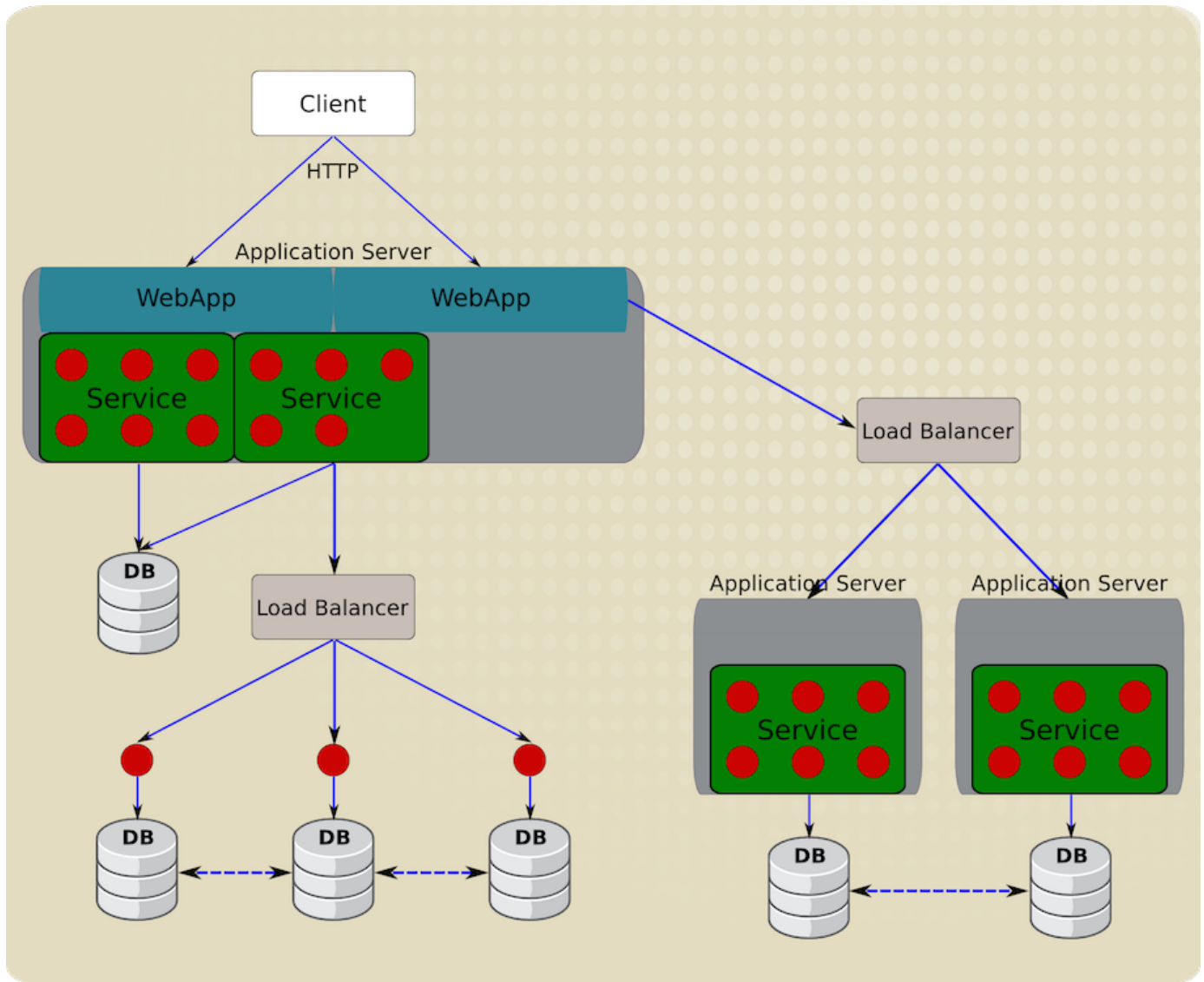


Figure 4. Tactical Microservices, HA

In the simple (and unrealistic) scenario that the remainder of the application requires a single instance while the business service needs two and the new microservice has three instances, the application directs its calls to load balancer, which in turn distribute the load between available services and provide the necessary failover.

The new services are isolated and the rest of the application is at least partially protected from their failure. These services may be scaled out dynamically as required without the overhead of replicating the entire environment.

2.3.4. Strategic Microservices

The architecture previously described in Tactical Microservices is either reactively separating out microservices that require complete isolation or have separate scaling needs, or anticipating such scenarios and proactively deploying them as individual microservices.

The Microservice Architecture paradigm can be fully embraced by decomposing entire applications into microservices and implementing entire systems as separately deployed microservices regardless of actual or anticipated isolation needs of individual services:

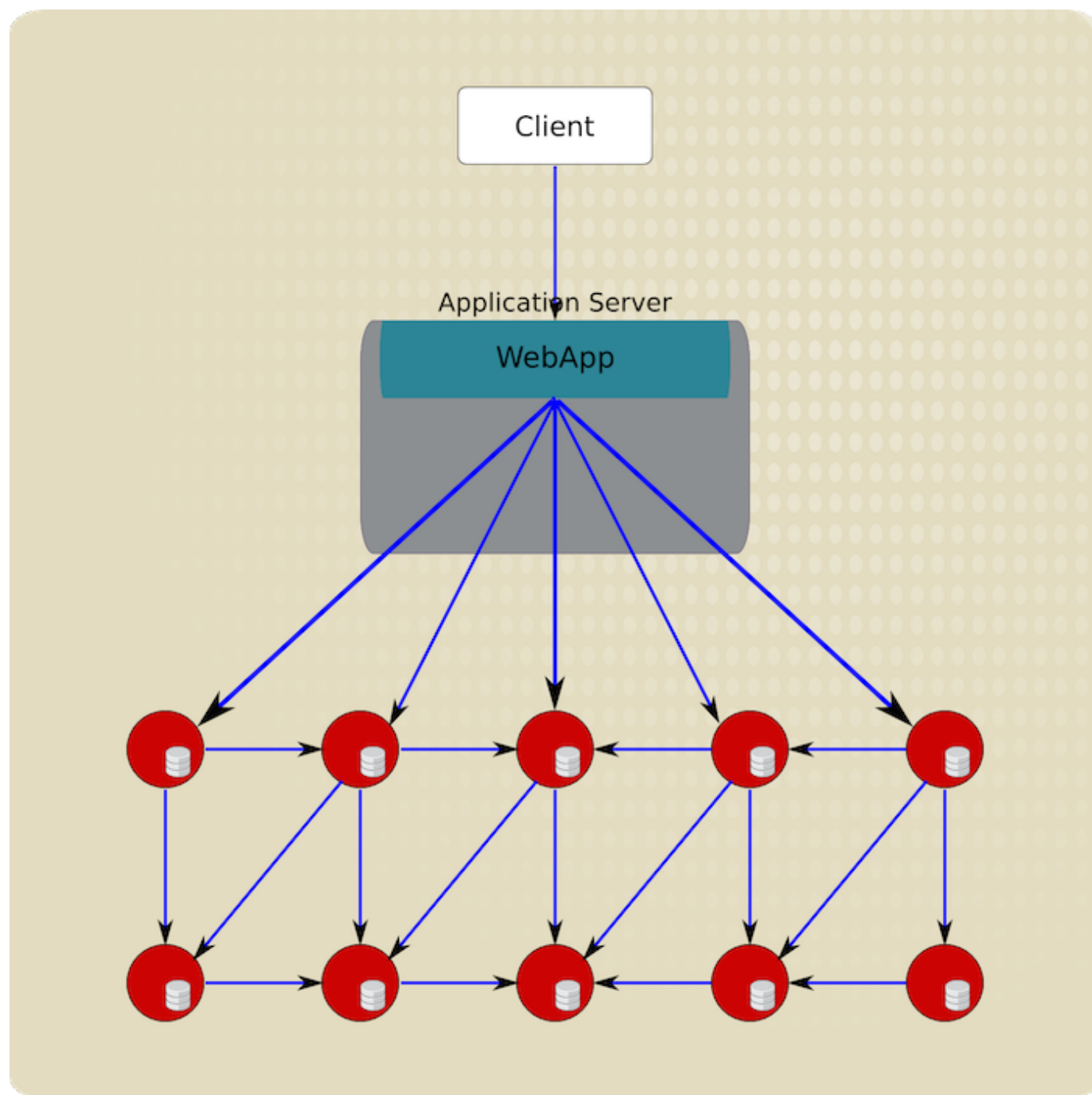


Figure 5. Strategic Microservices

In this architecture, each microservice includes its own persistence, which is at least logically encapsulated within the service. Each such service can be independently deployed, scaled, upgraded and replaced. The environment is fundamentally heterogeneous, so while frameworks and infrastructure services may be available to provide features and functions, each microservice is free to use its preferred technology. Some of these microservices may be running on a Java EE Server but overhead costs can be exorbitant and should be taken into consideration.

In this architecture, each microservice is easy to deploy, roll back and upgrade. Separate teams can work on separate microservices and the divide and conquer philosophy is used in full force. While the diagram depicts a single web application, there may in fact be zero, one, or many web applications invoking these microservices. Microservices may also depend on a data service layer for persistence, or may in fact not have any persistence requirements.

It is assumed that every microservice has multiple instances deployed but the number of instances depends on the load and mission criticality of the service in question. It is no longer necessary to deploy 10 copies of one service, because a different service needs 10 active copies to serve its purpose:

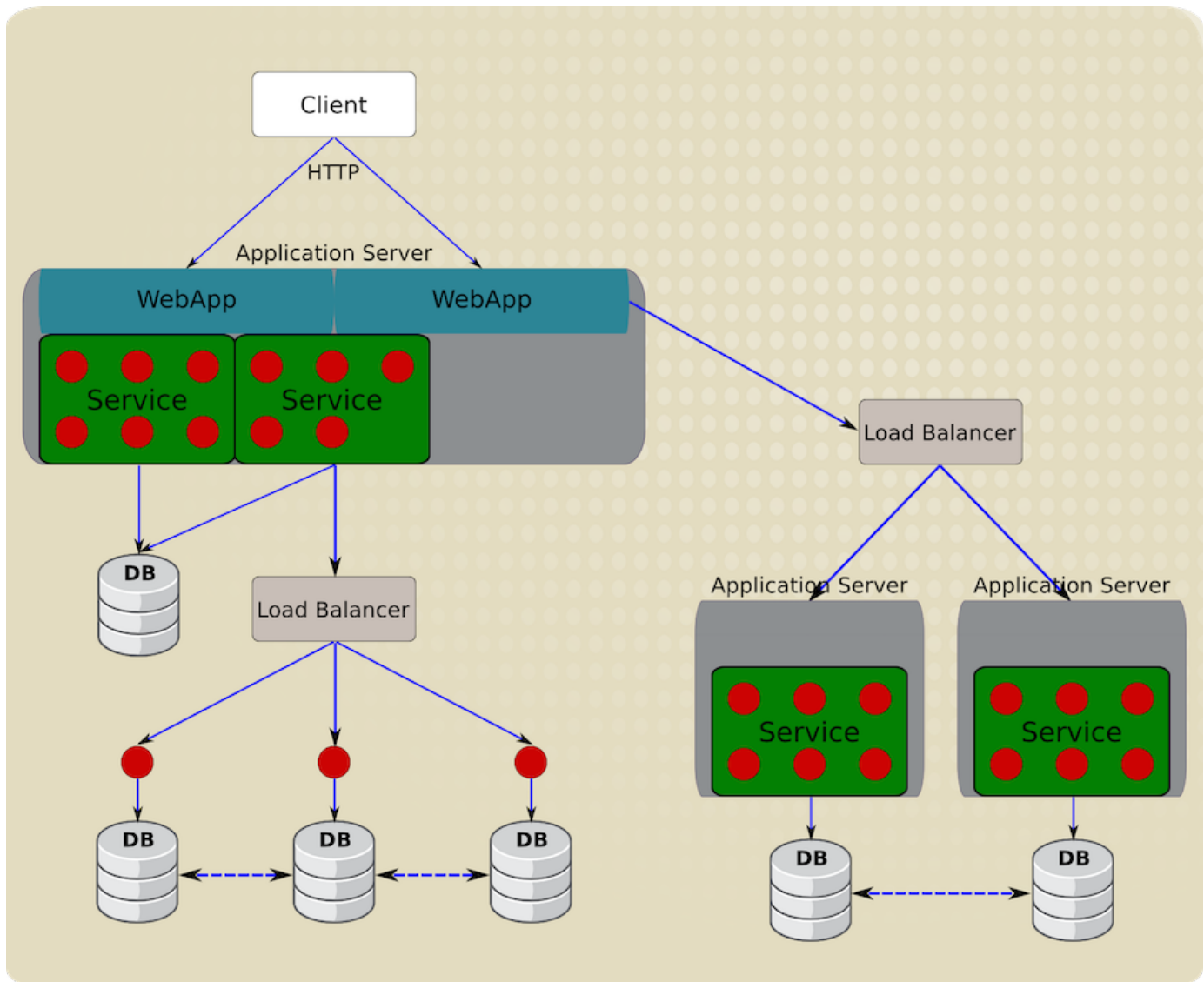


Figure 6. Strategic Microservices, HA

Notice that in this architecture diagram, some microservices are depicted as having fewer instances than others. Another obvious benefit of this approach is that the failure of a host due to a misbehaving service can be tolerated with minimal impact on other services.

As shown in the diagram, each microservice has its own persistence store. This is of course a logical data store to avoid creating dependency or coupling between the microservices. This same objective can be achieved by abstracting away the data store with a data service layer. This reference application makes a compromise in using a single database server, accessed directly by the microservices, while using separate schemas to segregate the services.

However, with a large number of microservices, each service may depend on a number of other services, each of which is deployed and scaled out in unknown locations. This leads to the requirement for a comprehensive service discovery solution, where services are registered as soon as they come up on a node and deregistered when they are taken offline. To avoid a single point of failure, such a service discovery solution would have to be replicated and highly available. Despite its HA nature, most services would also need to cache the results and be prepared to work when unable to access this solution.

Load balancing can quickly get more complex when a large number of microservices are scaled out in different numbers and the dependency graph gets more depth and breadth. Services might require their own distinct load balancing strategy and an extra hop in such an environment may prove costlier than usual.

The performance cost of making repeated remote calls typically leads to extensive caching requirements. The most often requirement is to have a service cache so that repeated and often expensive calls to the same microservice may be avoided.

High granularity along with a distributed deployment can also lead to orchestration challenges. Because of network latency, parallel invocation of services often becomes desirable, leading to the need for a queuing, asynchronous invocation and orchestration of requests and responses.

In general, as the environment becomes more heterogeneous, using uniform tooling and infrastructure becomes a less viable option.

2.3.5. Business-Driven Microservices

It must be emphasized that a microservice architectural style carries a lot of real benefits along with very real costs. The complexity of the system can grow exponentially with a large number of distributed components, each separately scaled out and perhaps dynamically auto-scaled.

Like most decisions, this does not have to be a binary choice. The modularity of the services can determine the complexity of the environment as well as the benefits and costs that are realized.

A distributed business-driven microservice architecture can achieve many of the benefits, while avoiding some of the costs:

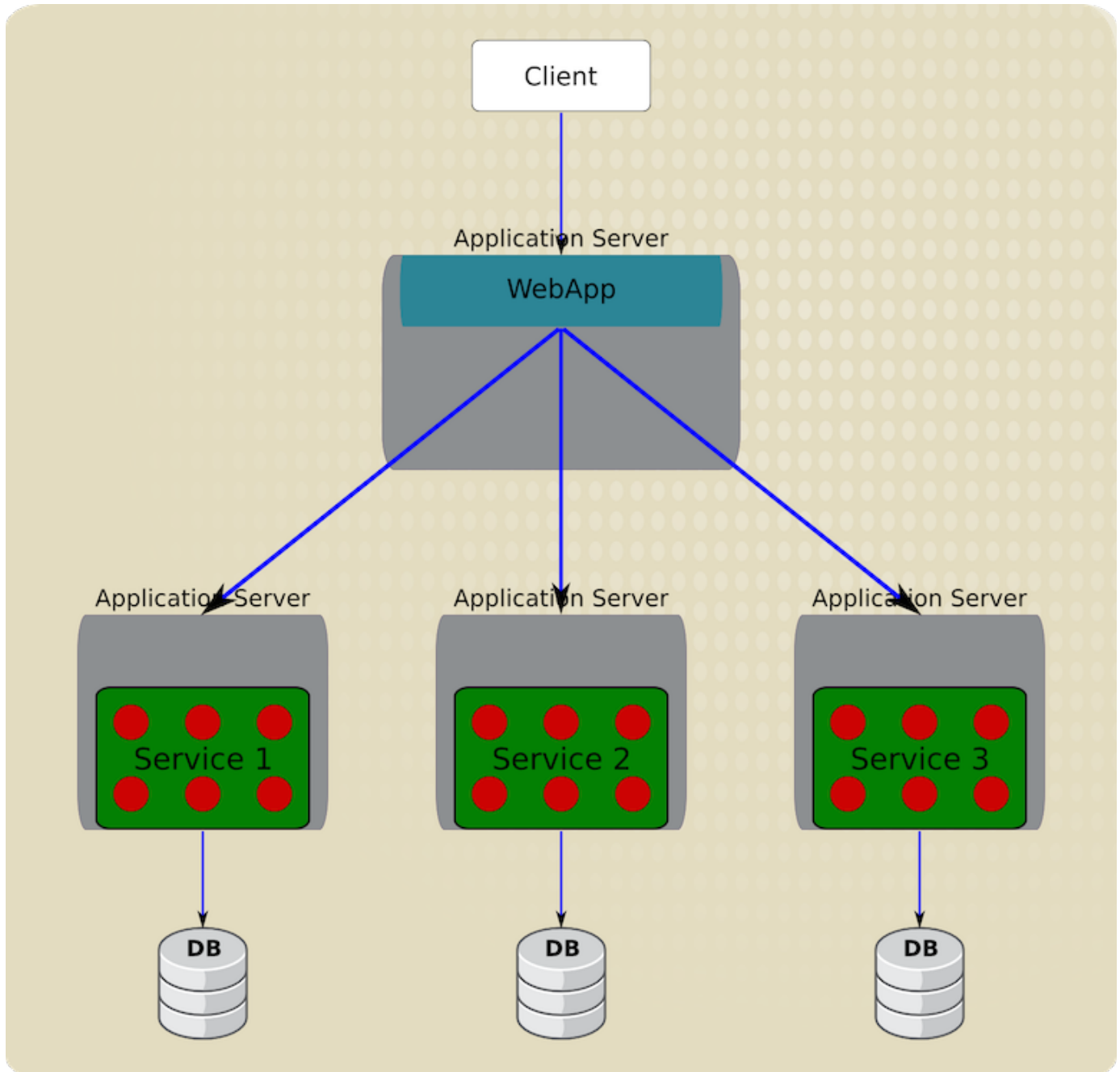


Figure 7. Business-Driven Microservices

An important and distinguishing characteristic of this architecture is that microservices do not communicate with one another. Instead, an aggregation layer is provided in the form of a web application that provides the required coordination.

The three services in this architecture diagram exist within a trust perimeter and the web application is the only client permitted to directly access these services. To use a different presentation technology, the web layer may be replaced with an aggregation layer that exposes a REST or other API. For example, JavaScript and similar technology may replace the Servlet layer and instead make direct calls to the server. In such a setup, the aggregation layer would be the only service exposed to outside clients and it would carefully design and expose an API where each operation would orchestrate and coordinate the underlying services as needed.

The architecture presented in this diagram avoids certain costs and continues to benefit from supported and familiar products and frameworks by constraining modularity and avoiding complex dependency graphs.

In its simplest form, microservices in this architecture remain self-contained within the system by avoiding any dependencies on other microservices. This does not include external dependencies, but is an attempt to simplify the environment by avoiding a large and deep dependency graph within the system.

When a certain component requires special consideration, either in its scaling requirements or in terms of fault isolation, it can be broken out and deployed independently. This can lead to a hybrid solution incorporating some of the tactical considerations of the previously described architecture depicted in [Tactical Microservices](#).

System requirements, willingness to be an early-adopter, in-house skill set, required agility and other factors can determine the best fit for an environment. There are systems and environments, for which a monolithic application architecture is the best fit. There are also very agile software groups creating fast-evolving systems that receive a large return on investment in strategic microservices. For a large group in-between, this approach can be a safe and rewarding compromise to gain many of the benefits without paying all of the costs.

Application server clustering can be used in this model to provide high availability. When employing horizontal scaling to provide redundancy and load balancing, service modularity determines the pieces that can be scaled out separately.

In [Business-Driven Microservices](#), Service 1 may be part of a 3-node cluster while Service 2 is clustered as 10 nodes and Service 3 only has a single active backup. Likewise, a catastrophic failure as a result of Service 1 would have no impact on Service 2 and Service 3, as they are separately packaged and deployed.

2.4. Cross-cutting concerns

2.4.1. Overview

Any system has a series of cross-cutting concerns, preferably addressed through consistent and common solutions that result in easier maintenance and lower cost. There are mature tools, frameworks and services that address such concerns and continue to be useful in various architectural styles.

The distributed and modular nature of microservice architecture creates new priorities and raises specific concerns that are not always adequately satisfied by traditional and available solutions.

The nature and specifics of these cross-cutting concerns will depend on the modularity of a microservice architecture. At one end of the spectrum, monolithic applications represent traditional enterprise applications that have been successfully operated in production environments for years and already benefit from a large array of established tools. At the other end of the spectrum, a highly modular and distributed MSA environment, described as strategic microservices in this document, introduces requirements that have not always existed in other architectures and do not have established and mature solutions.

While the concept of a microservices platform is not a well-defined industry term at the time of writing, it will inevitably emerge as the paradigm becomes more prevalent. Such a platform would provide value by filling in the missing pieces and ease the burden that is placed on early adopter today.

2.4.2. Containerization

An important feature and arguably the cornerstone of the microservice architecture is the isolated and individual deployment of each service. It is important that every instance of each microservice would have complete autonomy over the environment. Given the high granularity and the relatively small size of each service, dedicating a physical machine to each instance is not in consideration.

Virtualization reduces the overhead cost of each logical machine by sharing host resources and is often an acceptable environment for microservices. However, **Linux** containers and **Docker** technology in particular have improved on the benefits of virtualization by avoiding the full cost of an operating system and sharing some of host services that would be unnecessarily duplicated in each virtual machine.

Docker containers are emerging as the preferred units of deployment for microservices.

2.4.3. Service Discovery

Highly granular MSA environments typically involve dozens of services, each deployed as multiple instances. The dependency graph for some service invocations may involve as many as 10 to 20 calls and be up to 4 or 5 levels deep. This type of distribution makes a comprehensive service discovery solution critical. To take advantage of service redundancy, the caller needs to locate available and deployed instances of any given service at the required time.

The service discovery solution would have to include a distributed and highly available service registry where each service instance can register itself upon deployment and de-register on shutdown. There often needs to be a health check mechanism to remove instances that have suddenly dropped off, or be notified of failures to reach a service.

Communication with the service registry is best achieved through REST calls over HTTP to ensure that the solution remains language and platform agnostic.

Red Hat JBoss Data Grid provides a large number of features including RESTful interfaces, queries, customization and of course replication, that make it an attractive foundation for a service registry.

2.4.4. Load Balancer

One of the obvious costs of microservice architecture is the network latency that is introduced by the number of hops as a service dependency graph is traversed. Using a traditional load balancer typically doubles this latency by introducing an extra hop on each microservice invocation. For strategic microservices and in what is often an already chatty network environment, these extra hops are typically not acceptable. This architecture benefits from a load balancing solution that can be embedded in the client to eliminate the extra remote call. Such a framework would benefit from an IoC approach, allowing each caller to determine the load balancing strategy according to the circumstances.

2.4.5. Cache

In addition to common caching requirements in enterprise applications, typically used in front of databases or other remote and expensive calls, the distribution of functionality in an application often leads to repeated remote calls to a service, requesting the same information and unnecessarily increasing its load.

In microservice architecture environments with a large number of fine-grained microservices, it is prudent to identify those services that are often repeatedly called with the same request and take advantage of a service cache to increase performance and reduce resource cost.

Red Hat JBoss Data Grid provides a powerful caching solution with support for geographically distributed data, data sharding, consistent hashing algorithm and many other useful and relevant features that make it a great fit for an MSA environment.

2.4.6. Throttling, Circuit Breaker, Composable Asynchronous Execution

Complex dependency graphs along with network latency often make parallel invocation of services a necessity. To successfully orchestrate calls to dependencies while taking advantage of parallel execution, a sync to async pattern is often required. Once such an approach has been implemented, it becomes fairly easy to throttle calls to a service, or outbound calls from a service. The JAX-RS 2.0 Specification provides an implementation of asynchronous REST processing as well as REST clients.

Another critical design pattern for an MSA environment is the circuit breaker, which can limit the

number of threads stuck while attempting to call a single service and protect the rest of the environment from faulty services.

2.4.7. Security

Authentication and Authorization requirements are ubiquitous in practically all software environments.

One of the primary considerations in a microservice architecture environment is how the user identity will be propagated through the distributed service call. While many such environments may designate a security perimeter and not have each service be concerned with authenticating the end user, this approach is neither advisable nor acceptable in all situations.

Industry standards such as OAuth2, SAML and similar token-based security solutions provide a natural fit for RESTful services in a distributed environment. JBoss software provides support for these standards and satisfies associated security requirements through the **PicketLink** and **Keycloak** projects.

2.4.8. Monitoring and Management

The monitoring and management aspect of microservices are highly dependent on the deployment environment.

Most microservice deployments occur on an on-premise or public cloud environment. These cloud environment typically include native monitoring and management tools that can easily be used for the deployed services.

2.4.9. Resilience Testing

Microservices are designed and built to have the overall system withstand the failure of individual services. Like any feature or objective, this attribute needs to be tested and verified.

Test suites often need to be developed to verify the resilience of the system when unexpected load is placed on one service or a defect causes some service instances to break down.

Available testing and environment frameworks are often adapted to created the necessary QA tools for MSA environments.

2.5. Anatomy of a Microservice

The microservice architectural style lays out a set of principles on how application functionality can be decomposed into modular services, how these services should be deployed and the best practices around their inter-communication and other aspects of the architecture.

It is no coincidence that the design and development of the microservice itself is not part of this conversation. One of the stated goals of the microservice architectural style is to allow choice for the developers of each microservice to use the best tools and technologies, without the need to conform to an enterprise-wide or even a system-wide standard.

Despite this choice and the variety in both the requirements and their implementation from one service to another, these services largely resemble other enterprise software components. The term microservice may mislead some to view it as a trivial component but any system justifying the adoption of microservice architecture is complicated enough that each microservice will have its own significant dependencies and technical requirements.

Most microservices require persistence and need database connection pooling and connection management. Some have external dependencies and need to integrate with legacy system. Oftentimes, a microservice needs to enforce authentication and authorization; it would therefore benefit from declarative security. When a service performs several tasks as part of the same responsibility, even transactional behavior within the service may be required or beneficial.

These requirements are fundamentally no different than common enterprise software requirements that have led to the prevalence of application servers. The biggest impediment of using a Java EE application server to host an individual microservice is the resource usage and high fixed cost. Application servers are designed to act as shared infrastructure for a large number of software components and with enough load, the overhead cost is diminished in comparison. In microservice architecture where each service is deployed separately, this overhead can become prohibitively large.

JBoss EAP 7 benefits from an exceptional level of modularity afforded to the platform by the use of JBoss Modules. As a result, the platform can be configured to exclude modules that are not used by a given microservice and minimize the overhead.

While the ultimate choice of structure and deployment for each microservice is made separately, the benefits of creating a microservice as a JBoss EAP 7 application are well worth considering.

3. Reference Architecture Environment

This reference architecture mainly serves to demonstrate the distributed architecture of the application as multiple microservices and conversely, the composition of three microservices along with a presentation aggregation layer to form a functioning application.

The layers of this reference architecture roughly follow the one depicted in the [diagram](#) for Business-Driven Microservices.

The client, typically a web browser, makes calls to the load balancer layer. The load balancer is an **Apache HTTP Server** web server using **mod_proxy** to balance incoming requests between the three nodes of the next layer. Load balancing uses a simple round-robin algorithm with sticky behavior, ensuring that barring a server failure, a given user always reaches the same server node.

The second layer serves as both the aggregation and presentation layer and is the client to the microservices layer shown in the referenced architecture diagram. It is implemented as three separate logical machines, each hosting their own EAP instance and deploying the *presentation.war* application. With customer's shopping carts being persistent, the only in-memory state held by this application is the logged in user's identity. While in-memory state replication is offered by the EAP servers, the low impact of failure makes it optional. At worst and in the event of a server failure, the users with active sessions on that server will have to log in again to continue where they left off. This aggregation layer, mainly implemented in the *RestClient* class, makes its calls to the three microservices through the load balancer in the first layer.

The microservices layer consists of three sets of three logical machines, with each set hosting one of the microservices. These microservices are completely stateless and use simple load balancing by **mod_proxy** without any sticky session behavior.

The database layer consists of two logical database servers, one for the product and another for the sales microservice. In terms of the physical deployment, these two databases are implemented as two schemas of the same **MariaDB** server, hosted on the same physical machine as the Apache HTTP Server.

4. Creating the Environment

4.1. Prerequisites

Prerequisites for creating this reference architecture include a supported Operating System and JDK. Refer to [Red Hat documentation for supported environments](#).

With minor changes, almost any RDBMS may be used in lieu of **MySQL** Database, but if **MySQL** is used, the details of the download and installation are also considered a prerequisite for this reference architecture. On **RHEL 7**, use **MariaDB**, the community fork of **MySQL**:

```
# yum install mariadb-server
```

This reference architecture also uses **Apache HTTP Server** to demonstrate load balancing. In a production environment, consider using Red Hat JBoss Core Services Apache HTTP Server, but the functionality and configuration described in this document is largely similar for other versions of **httpd**. On a RHEL machine with the required Red Hat subscription, **Red Hat JBoss Core Services Apache HTTP Server 2.4** can be installed by using **yum**:

```
# yum groupinstall jbcs-httpd24
```

For further information on installing and configuring JBoss EAP Apache HTTP Server, refer to the JBoss EAP 7 Clustering reference architecture. Clients with access to the Red Hat Customer Portal may download the reference architecture and attachments from <https://access.redhat.com/site/articles/2359241>.

4.2. Downloads

The attachments to this document provide the Apache HTTP Server configuration file, CLI configuration to configure the required datasource, and the source code for the reference application. These files may be downloaded from:

<https://access.redhat.com/node/2386641/40/0>

If you do not have access to the Red Hat customer portal, See the Comments and Feedback section to contact us for alternative methods of access to these files.

Download JBoss EAP 7 from [Red Hat's Customer Support Portal](#):

- Red Hat JBoss Enterprise Application Platform 7.0.0

An appropriate version of MySQL JDBC driver should also be separately downloaded.

4.3. Installation

Red Hat's JBoss EAP 7 does not require any installation steps. The archive file simply needs to be extracted after the download:

```
# unzip jboss-eap-7.0.0.zip
```

Place the EAP files in an appropriate location, for example: */opt/jboss-eap-7.0*

4.4. Configuration

This reference architecture uses **firewalld**, the default Red Hat Firewall, to block all network packets by default and only allow configured ports and addresses to communicate. Refer to the Red Hat documentation on **firewalld** for further details on this tool.

Check the status of firewalld on each machine and make sure it is running:

```
# systemctl status firewalld
```

This reference environment starts with the default and most restrictive firewall setting and only opens the required ports. In particular, the machine hosting the Apache HTTP Server load balancer needs to open port 80 to all incoming traffic:

```
# firewall-cmd --zone=public --add-service=http --permanent
```

The machine hosting the database must allow connections to the database port, in this case 3306, from the hosts where microservices are deployed. In this reference environment, these boxes use a sequence of IP addresses in the same subnet:

```
# firewall-cmd --permanent --zone=public --add-rich-rule="rule family='ipv4' source address='10.19.137.1/24' port protocol='tcp' port='3306' accept"
```

Using the permanent flag persists the firewall configuration but also requires a reload to have the changes take effect immediately:

```
# firewall-cmd --reload
```

On each of the hosts serving a microservice, open the JBoss EAP port for incoming traffic from the load balancer. In the reference environment, the Apache HTTP Server load balancer is hosted at 10.19.137.30. Follow that up by reloading firewall configuration to have the change take effect:

```
# firewall-cmd --permanent --zone=public --add-rich-rule="rule family="ipv4" source
address="10.19.137.30/32" port protocol="tcp" port="8080" accept"
# firewall-cmd --reload
```

This reference environment has been set up and tested with **Security-Enhanced Linux (SELinux)** enabled in *enforcing* mode. Once again, refer to the Red Hat documentation on SELinux for further details on using and configuring this [feature](#). For any other operating system, consult the respective documentation for security and firewall solutions to ensure that maximum security is maintained while the ports required by your application are opened.

When enabled in *enforcing* mode, by default, SELinux prevents Apache HTTP Server from establishing network connections. On the machine hosting Apache HTTP Server, configure SELinux it to allow httpd network connections:

```
# /usr/sbin/setsebool httpd_can_network_connect 1
```

This reference application uses Apache HTTP Server to host static content, in this case the images used by the presentation layer. Create the images directory on the machine hosting httpd, copy the images in there, and set security privileges as appropriate to allow them to be served:

```
# mkdir -p /srv/msa/images
# cp code/Presentation/images/* /srv/msa/images/
# chmod 644 /srv/msa/images/*
```

Even with the correct security privileges, SELinux can stop the images from being served to web users unless their extended attributes are properly set:

```
# chcon -R -t httpd_sys_content_t /srv/msa/images
```

Various other types of configuration may be required for UDP and TCP communication. For example, **Linux** operating systems typically have a low maximum socket buffer size configured, which is lower than the default cluster **JGroups** buffer size. It may be important to correct any such warnings observed in the EAP logs. For example, in this case for a **Linux** operating system, the maximum socket buffer size may be configured as follows.

```
# sysctl -w net.core.rmem_max=26214400
# sysctl -w net.core.wmem_max=1048576
```

The reference application uses a series of host names to make it more portable. These hostnames are as follows:

- product-service: Load balancer front-ending product microservice nodes
- billing-service: Load balancer front-ending billing microservice nodes
- sales-service: Load balancer front-ending sales microservice nodes
- product-db: The location of the product database server
- sales-db: The location of the sales database server

In the reference environment, the machine addressed 10.19.137.30 hosts the Apache HTTP Server, effectively acting as the load balancer for all three microservices. This machine also hosts the single database server with two separate schemas for sales and product, acting as two logical database servers.

For testing purposes, it is easy enough to edit the hosts file on each of the thirteen machines (three nodes for each of presentation, product, sales and billing applications, plus one for database and load balancer):

```
# vi /etc/hosts
```

Add the following lines to the *hosts* file:

```
10.19.137.30 product-service
10.19.137.30 billing-service
10.19.137.30 sales-service
10.19.137.30 product-db
10.19.137.30 sales-db
```

4.4.1. Red Hat JBoss Core Services Apache HTTP Server

The Apache HTTP Server used in this reference environment serves as the load balancer for the presentation application as well three different microservices. This web server also serves the static content (images) for the presentation layer.

Create a virtual host for the presentation layer. This environment uses msa-web as the host name for the entry point:

```
<VirtualHost msa-web:80>
    ProxyPreserveHost On
    ProxyRequests off

    ServerName msa-web
```

Configure the location of static resources and allow access:

```
# static files:
DocumentRoot /srv/msa/

<Directory /srv/msa>
    Options All
    AllowOverride All
    Require all granted
</Directory>
```

Finally, configure **mod_proxy** as a load balancer to distribute load between the nodes of the presentation application running on JBoss EAP.

```
<Proxy balancer://presCluster>
    BalancerMember http://10.19.137.31:8080
    BalancerMember http://10.19.137.32:8080
    BalancerMember http://10.19.137.33:8080

    Order Deny,Allow
    Deny from none
    Allow from all

    ProxySet lbmethod=byrequests
</Proxy>

ProxyPass /presentation balancer://presCluster/presentation
stickysession=JSESSIONID|jsessionid scolonpathdelim=On
ProxyPassReverse /presentation balancer://presCluster/presentation
</VirtualHost>
```

The load balancing method is set to byrequests to perform a default simple round-robin, but sticky sessions are turned on using either the cookie name or URL rewriting with semicolon as a separator.

The load balancer configuration for the the product microservice is much simpler, since no static

resources are being served and the application is completely stateless, so sticky sessions are not applicable:

```
<VirtualHost product-service:80>
  ProxyPreserveHost On
  ProxyRequests off

  ServerName product-service

  <Proxy balancer://productCluster>
    BalancerMember http://10.19.137.34:8080
    BalancerMember http://10.19.137.35:8080
    BalancerMember http://10.19.137.36:8080

    Order Deny,Allow
    Deny from none
    Allow from all

    ProxySet lbmethod=byrequests
  </Proxy>

  ProxyPass /product balancer://productCluster/product
  ProxyPassReverse /product balancer://productCluster/product
</VirtualHost>
```

The load balancer configuration for the sales and billing microservices is similar:

```
<VirtualHost sales-service:80>
  ProxyPreserveHost On
  ProxyRequests off
  ServerName sales-service

  <Proxy balancer://salesCluster>
    BalancerMember http://10.19.137.37:8080
    BalancerMember http://10.19.137.38:8080
    BalancerMember http://10.19.137.39:8080
    Order Deny,Allow
    Deny from none
    Allow from all
    ProxySet lbmethod=byrequests
  </Proxy>

  ProxyPass /sales balancer://salesCluster/sales
  ProxyPassReverse /sales balancer://salesCluster/sales
</VirtualHost>

<VirtualHost billing-service:80>
  ProxyPreserveHost On
  ProxyRequests off

  ServerName billing-service

  <Proxy balancer://billingCluster>
    BalancerMember http://10.19.137.40:8080
    BalancerMember http://10.19.137.41:8080
    BalancerMember http://10.19.137.42:8080

    Order Deny,Allow
    Deny from none
    Allow from all

    ProxySet lbmethod=byrequests
  </Proxy>

  ProxyPass /billing balancer://billingCluster/billing
  ProxyPassReverse /billing balancer://billingCluster/billing
</VirtualHost>
```

Restart the web server after configuration is complete:

```
# systemctl stop jboss-httpd24-httpd.service
# systemctl start jboss-httpd24-httpd.service
```

4.4.2. MySQL / MariaDB Database

Start the database server:

```
# systemctl start mariadb.service
```

Enable the database server at boot time:

```
# systemctl enable mariadb.service
```

Initialize the database by running the included script:

```
# mysql_secure_installation
```

The database root password is initially blank. Set a new password and remove anonymous users and the test database, before reloading the privilege tables.

Once the database initialization is complete, run the database utility to set up the application databases:

```
# mysql -u root -p
```

Log in using the newly configured password and use MySQL DDL syntax to create the database and the user that accesses it:

```
CREATE DATABASE product;
USE product;
CREATE USER 'product'@'%' IDENTIFIED BY 'password';
GRANT USAGE ON . TO 'product'@'%' IDENTIFIED BY 'password';
```

Create tables along with the sequence used by JPA, for example:

```
CREATE TABLE Product (SKU BIGINT NOT NULL AUTO_INCREMENT, DESCRIPTION VARCHAR(255),  
HEIGHT NUMERIC(8,2) NOT NULL, LENGTH NUMERIC(8,2) NOT NULL, NAME VARCHAR(255), WEIGHT  
NUMERIC(8,2) NOT NULL, WIDTH NUMERIC(8,2) NOT NULL, FEATURED BOOLEAN, AVAILABILITY  
INTEGER NOT NULL, IMAGE VARCHAR(255), PRICE NUMERIC(9,2) NOT NULL, PRIMARY KEY (SKU))  
AUTO_INCREMENT = 10001;
```

Complete the setup of the databases by running the instructions in the provided SQL script file:
setup.sql

4.4.3. JBoss Enterprise Application Platform

This reference architecture makes very few changes to the JBoss EAP configuration. The only required change is to configure a datasource and connect it to the appropriate database for the *Product* and *Sales* server instances. To do this, the MySQL JDBC driver is installed as a module and configured on the server. The appropriate *datasource* is then configured to leverage this driver and connect to the corresponding database.

This reference architecture provides a JBoss CLI script to automate the required configuration. Sample property files have been provided for each of the *product* and *sales* datasources. Modify the *mysql.jar* property to indicate the exact name and version of the MySQL JDBC driver you downloaded. Also configure the username and password appropriately for your database. The JNDI name of the datasource is used by the application code and expected to match the provided value for each of the *product* and *sales* datasources.

With the correct property file configured for an environment and the JDBC driver downloaded and placed next to it, use JBoss CLI to configure the standalone server:

```
# /opt/jboss-eap-7.0/bin/jboss-cli.sh --file=configure.cli  
--properties=product.properties
```

Optionally, inspect the datasources subsystem of the server configuration for the applied changes in */opt/jboss-eap-7.0/standalone/configuration/standalone.xml*.

The JBoss EAP server can be started after this step and bound to the correct IP address to be accessible by the load balancer.

4.5. Deployment

The application code for the presentation layer and all three microservice projects is included under the *code* directory of the attachment.

At the top level of this directory is an aggregation POM that builds all four projects. To build the projects, run maven from this directory:

```
# cd code/  
# mvn install
```

Once the build completes successfully, the deployable web archive files are generated and placed in the target directory of each project. Deploy each web application by dropping it into the */opt/jboss-eap-7.0/standalone/deployments/* folder of its respective servers. The web application archives are as follows:

- The presentation layer: *code/Presentation/target/presentation.war*
- The product microservice: *code/Product/target/product.war*
- The sales microservice: *code/Sales/target/sales.war*
- The billing microservice: *code/Billing/target/billing.war*

While deploying the application, monitor the server log and make sure there are no errors. The server log is located at */opt/jboss-eap-7.0/standalone/log/server.log*

4.6. Execution

Open a browser and point it to the load balancer to reach the application. Assuming that the hostname for the load balancer machine is msa-web, point the browser to: <http://msa-web/presentation/>

The first request to the server returns the featured products from the database:

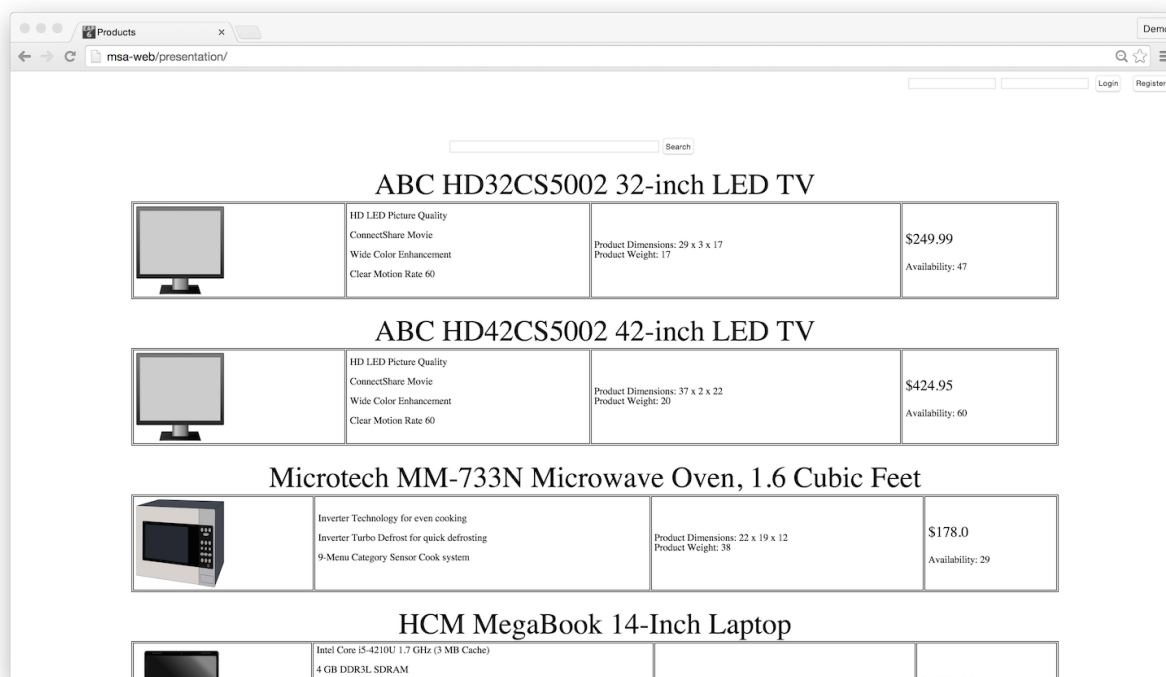


Figure 8. Featured Products

The provided search bar allows the user to filter the products by keyword. Each product in the database may be associated with one or multiple keywords. The scope of this search is all products, whether they are featured or not. Search the product database for TV:

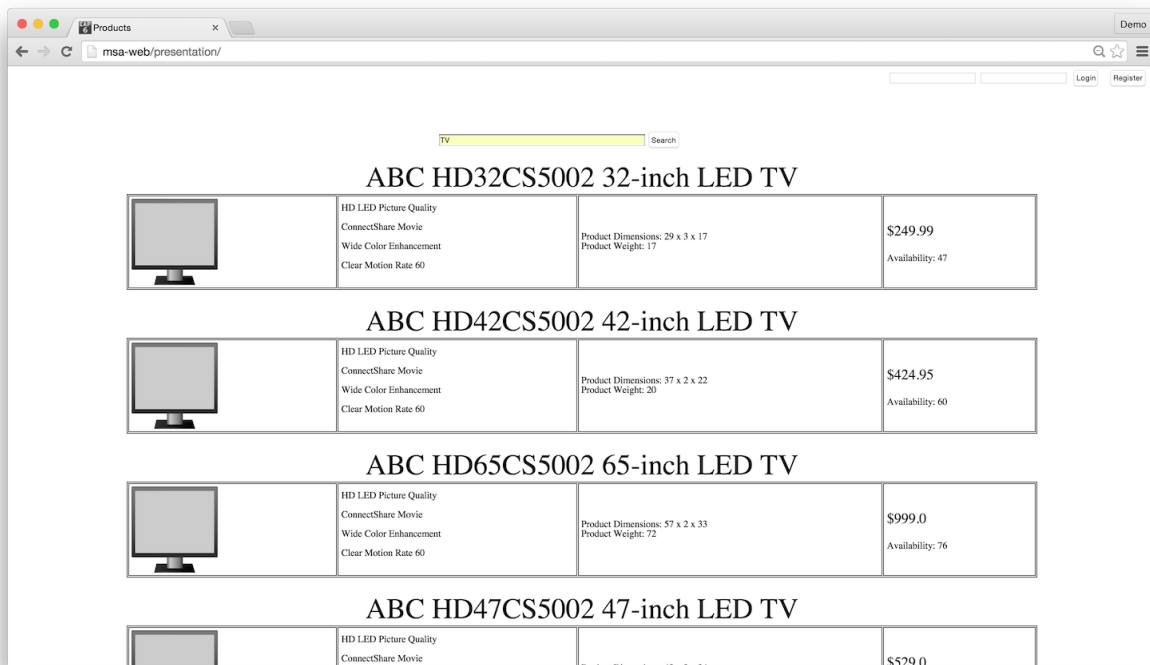


Figure 9. Search Products

Click the *Register* button on the top right of the screen to register a new customer user:

Customer Registration

Name:	Babak Mozaffari
Address:	12300 Wilshire, Los Angeles
Telephone:	310-123-1234
Email:	babak@redhat.com
Username:	Babak
Password:	••••••••

Figure 10. Customer Registration

Once registered, the user will be automatically and implicitly signed in. Alternatively, in a new browser session, the user can enter the same username and password to render the same page.

This page shows the same featured products but notice that there is now a purchase button underneath the product price and availability. Click on purchase button for a product to add it to your shopping cart:

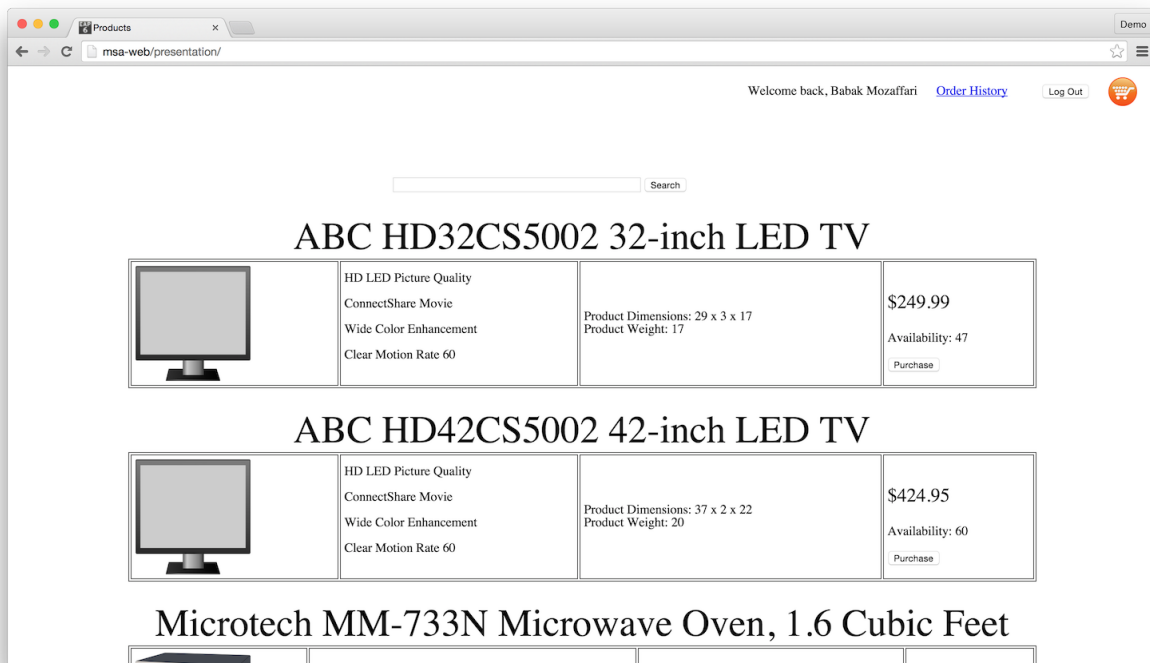


Figure 11. User logged in

Notice that the shopping cart icon on the top right of the screen now includes the number of items in the shopping cart:

Welcome back, Babak Mozaffari

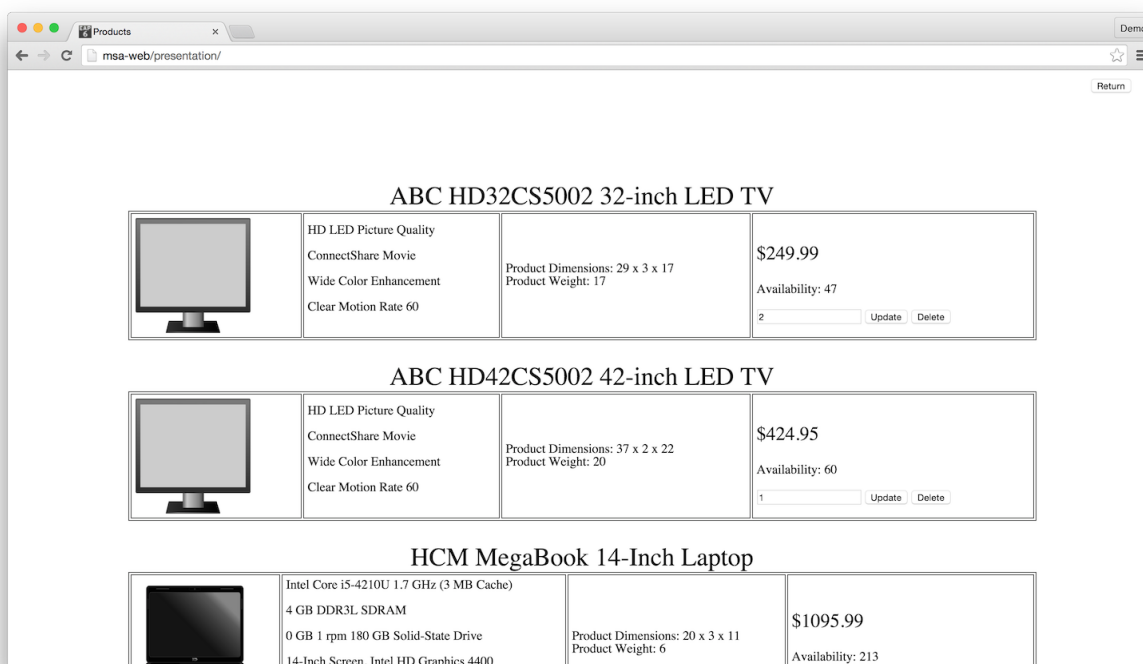
[Order History](#)

[Log Out](#)



Figure 12. Shopping Cart Item Count

To view the content of your shopping cart, click on the cart icon:



Shopping Cart			
Return			
ABC HD32CS5002 32-inch LED TV			
	HD LED Picture Quality ConnectShare Movie Wide Color Enhancement Clear Motion Rate 60	Product Dimensions: 29 x 3 x 17 Product Weight: 17	\$249.99 Availability: 47 2 Update Delete
ABC HD42CS5002 42-inch LED TV			
	HD LED Picture Quality ConnectShare Movie Wide Color Enhancement Clear Motion Rate 60	Product Dimensions: 37 x 2 x 22 Product Weight: 20	\$424.95 Availability: 60 1 Update Delete
HCM MegaBook 14-Inch Laptop			
	Intel Core i5-4210U 1.7 GHz (3 MB Cache) 4 GB DDR3L SDRAM 0 GB 1 rpm 180 GB Solid-State Drive 14-Inch Screen, Intel HD Graphics 4400	Product Dimensions: 20 x 3 x 11 Product Weight: 6	\$1095.99 Availability: 213

Figure 13. Shopping Cart Content

Scroll down and click the checkout button to pay for the items in the cart:

Order Summary

Product	Unit Price	Quantity	Product Cost
ABC HD32CS5002 32-inch LED TV	\$249.99	\$2.00	\$499.98
ABC HD42CS5002 42-inch LED TV	\$424.95	\$1.00	\$424.95
HCM MegaBook 14-Inch Laptop	\$1,095.99	\$1.00	\$1,095.99

Grand Total: **\$2,020.92**

Customer: Babak Mozaffari
Telephone: 310-123-1234
Address: 12300 Wilshire, Los Angeles
Credit Card No: 1234123412341234
Expiration Date: December 2015
Verification Code: 314

Submit

Cancel

Figure 14. Checkout

After entering valid data and an expiration data that is in the future, click the submit button to process the purchase.

Once the purchase is completed, the application returns to the featured products page. From this homepage, click the *Order history* link to view all the orders. The items in your shopping cart are stored as an order that is in progress and has a status of *Initial*:

Order 100049, Initial

Product	Unit Price	Quantity	Product Cost
ABC HD32CS5002 32-inch LED TV	\$249.99	\$3.00	\$749.97
ABC HD42CS5002 42-inch LED TV	\$424.95	\$1.00	\$424.95

Grand Total: **\$1,174.92**

Order 100048, Paid

Product	Unit Price	Quantity	Product Cost
ABC HD32CS5002 32-inch LED TV	\$249.99	\$2.00	\$499.98
ABC HD42CS5002 42-inch LED TV	\$424.95	\$1.00	\$424.95
HCM MegaBook 14-Inch Laptop	\$1,095.99	\$1.00	\$1,095.99

Grand Total: **\$2,020.92**

Transaction Number: 3360915

Transaction Date: Apr 2, 2015 7:05:41 PM

[Return](#)

Figure 15. Customer Order History

5. Design and Development

5.1. Overview

This section performs a step by step walkthrough of the design and development of the reference architecture application. A varying amount of focus is placed on different components. For example, the database is not a focus of this reference architecture and while the required SQL scripts are provided and described, the topic is not approached with a similar level of depth as other components. Similarly, the presentation layer developed using JSP technology is merely provided to demonstrate application functionality and is not a major focus of this reference architecture.

5.2. Integrated Development Environment

This reference architecture uses JBoss Fuse IDE plugins for JBoss Developer Studio 9.1.

5.2.1. JBoss Developer Studio

Download the *Stand-alone installer* for **JBoss Developer Studio (JBDS) 9.1.0** from the [Red Hat Customer Support Portal](#).

The installer is an executable JAR file. Installing a recent version of the JDK and having the java and associated commands in the execution path is a prerequisite to using JBDS and JBoss Fuse itself.

In most operating systems, it is enough to simply double-click the JBoss Developer Studio installation JAR file to start installing the IDE. You can also trigger the installation from the command line:

```
$ java -jar jboss-devstudio-9.1.0.GA-installer-standalone.jar
```

Accept the license, choose a location to install the product and proceed with the installation. Select the default or preferred JDK location. Is it not necessary to configure any platform or server location while installing JBoss Developer Studio.

Start JBoss Developer Studio by locating the shortcut created in the designated location. Select a location for the IDE workspace. Once started, an initial welcome screen appears. Close this screen to enter the familiar Eclipse framework environment.

5.2.2. Creating a Maven Project

Click the drop-down for the *New* toolbar icon at the top left of the JBoss Developer Studio window and select *Maven Project*. Alternatively, you can click the icon to open the *New* wizard dialog, open the group called *Maven* and select *Maven Project* from there.

Create a new project called *product* that will handle the definition and inventory management of the products sold through the e-commerce site that is the subject of this reference architecture's sample application.

The new project wizard prompts you to select a location for the project or use the default workspace location. For temporary and testing purposes, it is easiest to let JBDS simply create the project in the designated workspace. Select *Next* and choose the *jboss javaee6 blank webapp archetype* to create the basic structure for a Web Application project based on Maven. When a webapp archetype is available for a more recent and better matching version of JBoss EAP, it can be substituted, but the basic structure of the project remains the same:

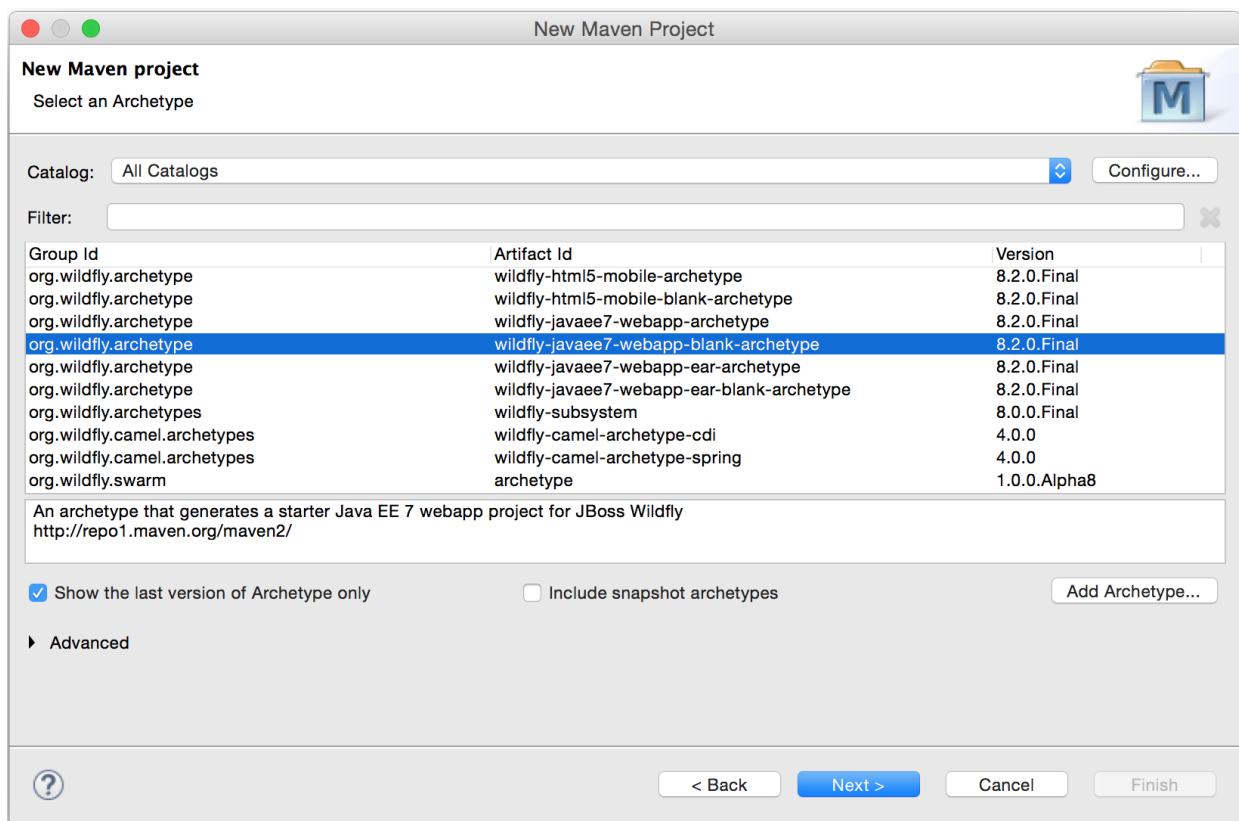
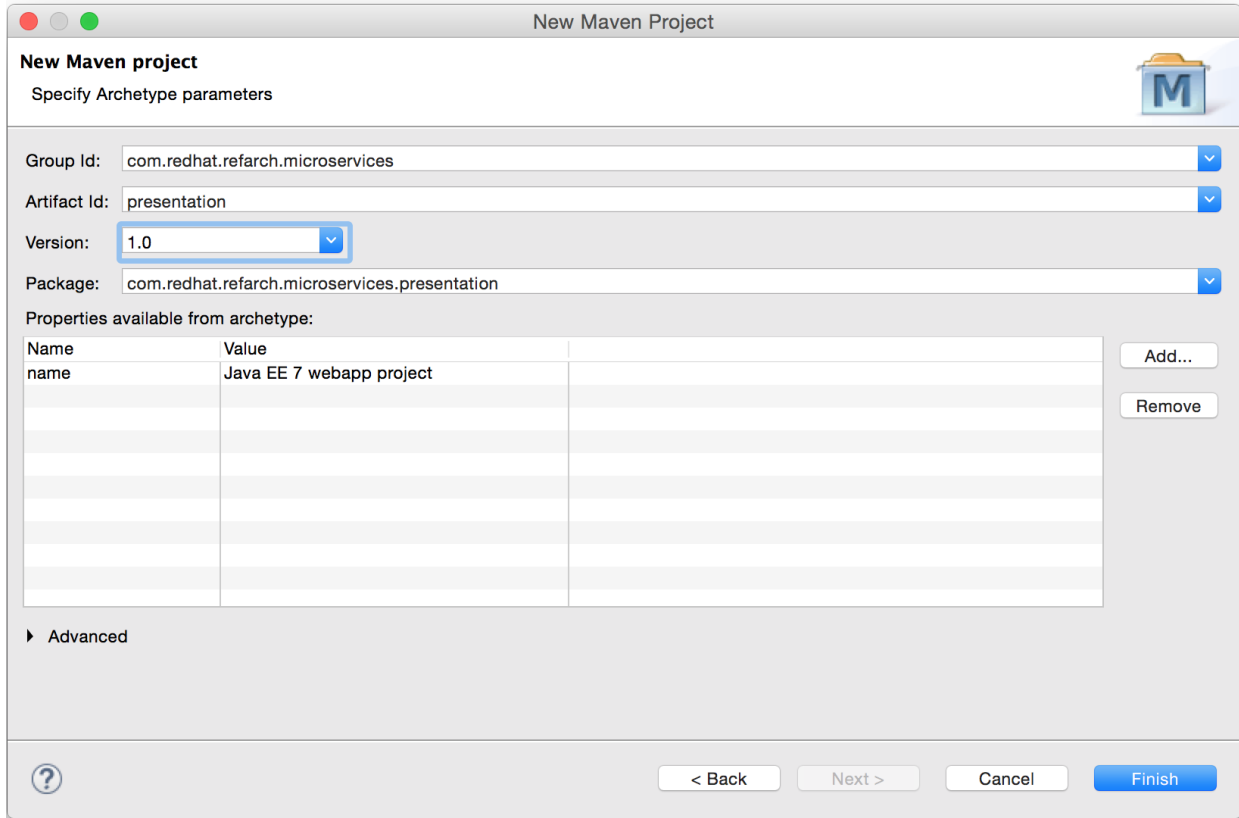


Figure 16. Maven Archetype

Use the next dialog to set the group and artifact Id of the project, as well as the version:



New Maven project
Specify Archetype parameters

Group Id:

Artifact Id:

Version:

Package:

Properties available from archetype:

Name	Value
name	Java EE 7 webapp project

Advanced

< Back Next > Cancel Finish

Figure 17. Maven Project

Click the finish button to complete the initial project setup.

5.2.3. Configuring Java 8

The project template is only a starting point and requires a number of additions, removals and modifications to adapt to individual projects. This reference architecture uses and relies on Java 8. Open the generated *Maven Project Object Model (pom)* file in JBoss Developer Studio and change the Java version:

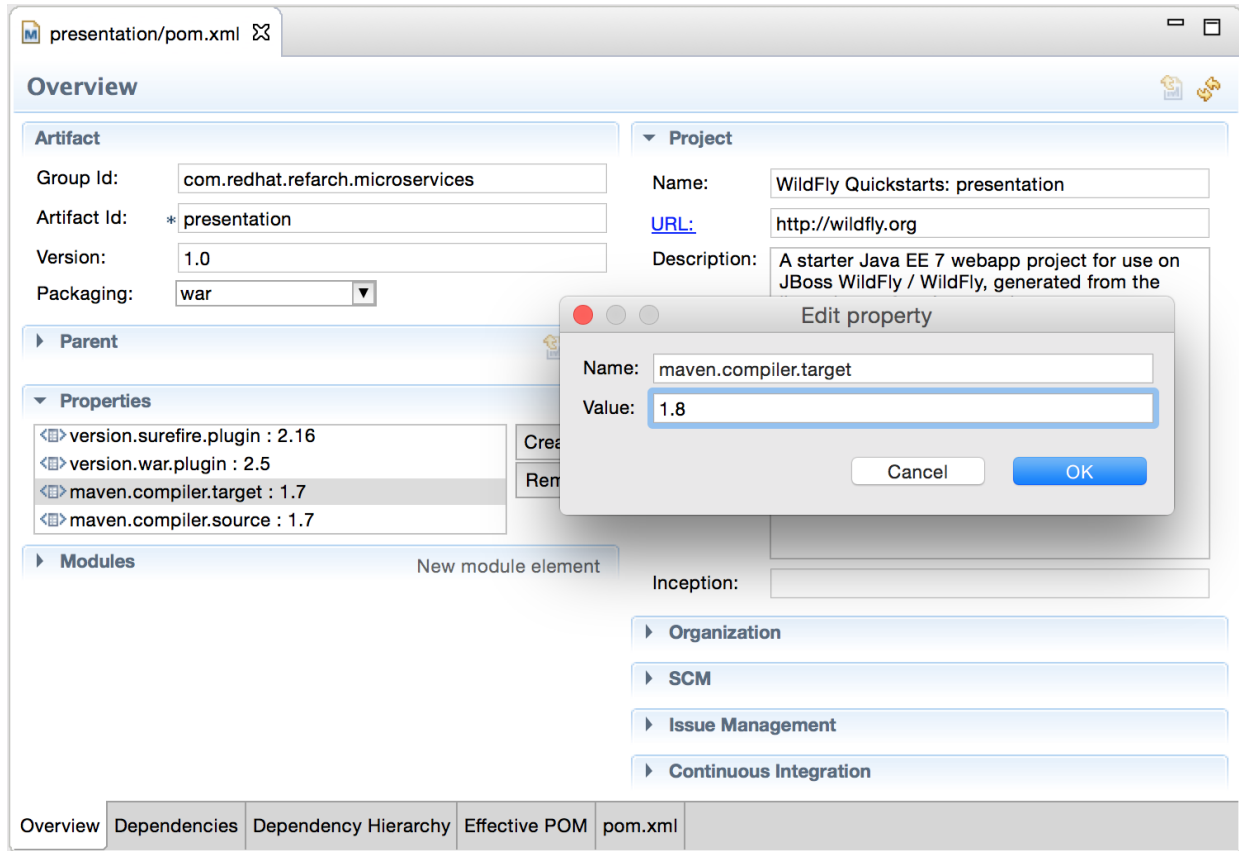


Figure 18. Java version in pom file

Change the Java version for both the source code and the generated artifacts to 1.8 by modifying the value of the *maven.compiler.source* and *maven.compiler.target* properties. In each case, double-click the property in the Overview window to open a dialog and edit the value.

5.3. Java Persistence API (JPA)

5.3.1. Overview

The maven template includes support for JPA and creates a default persistence configuration file to connect to a datasource.

5.3.2. Persistence Unit

This persistence configuration file is located at: `src/main/resources/persistence.xml`

As part of the maven template, this file includes a single persistence unit called *primary*. The name of the transactional datasource is derived from the project name. Review this configuration by opening the persistence xml file in JBoss Developer Studio and navigating to the *Connection* tab:

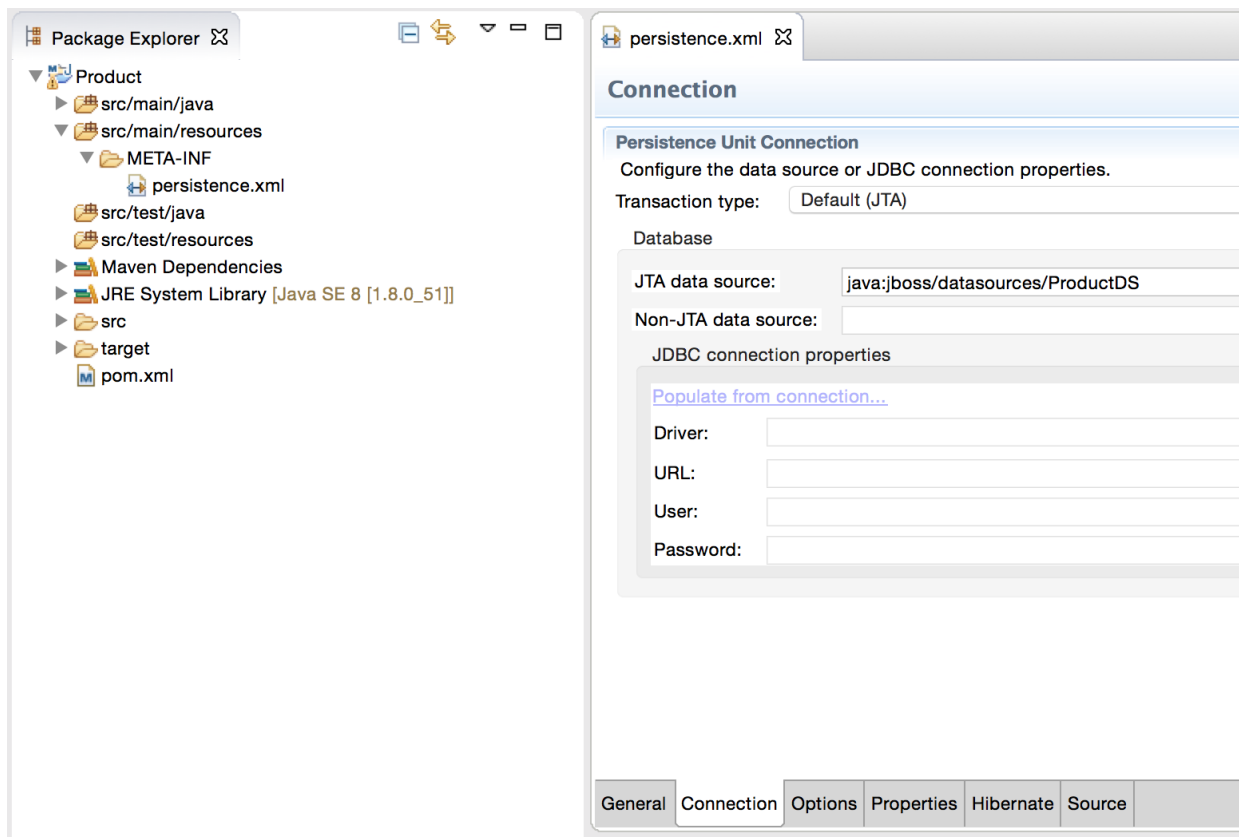
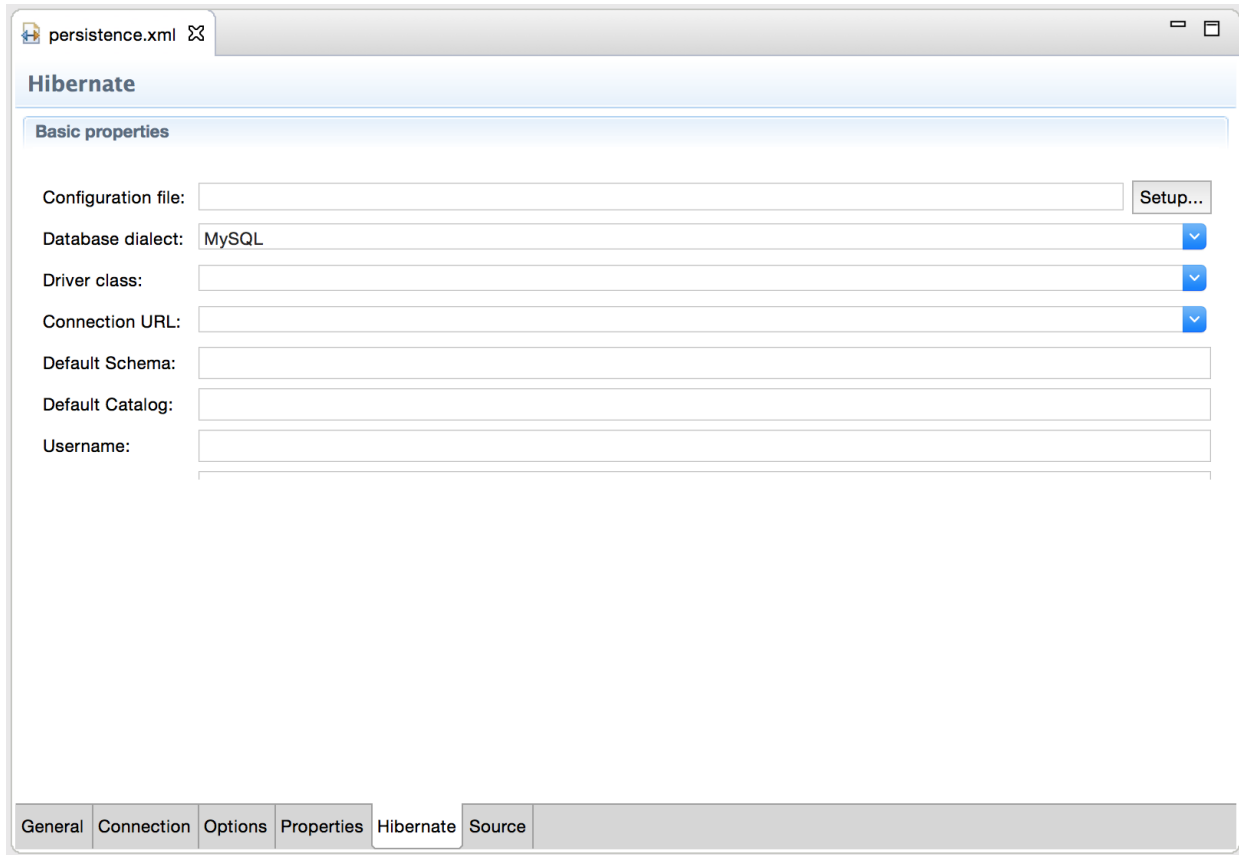


Figure 19. Datasource Configuration

Rename the data source to simply use a capital letter at the beginning: *ProductDS*

Change to the *Hibernate* tab. Configure hibernate to use your database of choice. This reference architecture uses **MySQL** and sets the Database *dialect* accordingly:



The screenshot shows an IDE window titled 'persistence.xml'. The 'Hibernate' tab is active. The 'Basic properties' section contains the following fields:

- Configuration file: [Empty text box] Setup...
- Database dialect: MySQL (dropdown menu)
- Driver class: [Empty text box]
- Connection URL: [Empty text box]
- Default Schema: [Empty text box]
- Default Catalog: [Empty text box]
- Username: [Empty text box]

At the bottom of the window, there are tabs for 'General', 'Connection', 'Options', 'Properties', 'Hibernate', and 'Source'. The 'Hibernate' tab is currently selected.

Figure 20. Hibernate Dialect Setting

For learning purposes, it can also be useful to turn on hibernate logging of SQL statements by setting *hibernate.show_sql* to true. This can be configured from the *Properties* tab.

Also note the *hibernate.hbm2ddl.auto* property, concerning the mapping of JPA classes to database tables. The template default value of create-drop results in non-persistent behavior between application redeployment and server restarts. This property should be set to a value of validate, or complete removed, when not in an early testing phase.

The remainder of this document assumes that this property has been removed. Instead, database scripts are used to create the required tables.

5.3.3. Persistence Entity

Create a JPA entity to represent a product that will be listed and sold through the e-commerce application.

JavaBean

Start by creating a simple JavaBean with the required properties:

- Long sku: Product SKU and a unique identifier for the product
- String name: User friendly name and identifier of the product
- String description: Full description of the product
- Integer length: Product dimensions, length in inches
- Integer width: Product dimensions, width in inches
- Integer height: Product dimensions, height in inches
- Integer weight: Product weight in pounds
- Boolean featured: Flag to indicate if the product should be featured on the homepage
- Integer availability: Inventory, available units of the product for sale
- BigDecimal price: Sale price in US dollars
- String image: Partial path to the product image on file or content management system

Generate getter and setter methods for these properties by using the corresponding action in the *Source* menu of JBoss Developer Studio. Make sure all the fields are selected before generating the methods.

From the same menu, generate equals and *hashCode* methods for this JavaBean. This time, only select the *sku* field and exclude all other fields. The SKU uniquely and distinctly identifies the product and can be used to determine if any two objects represent the same product or not.

Optionally, create a *toString* method for the JavaBean from the same menu to help log and troubleshoot the application. Select all the bean properties for this action.

These steps produce a well-designed and standard JavaBean class that can be used for a wide variety of purposes. Persistence entities provide Object-Relational Mapping (ORM) by using a Java class to represent a database table, where each instance of the class represents a row in the said table.

JPA Entity

Mark this JavaBean as a JPA bean by annotating the class with *javax.persistence.Entity*:

```
@Entity
public class Product
{
```

Primary key

The product SKU is a perfect fit as the primary key of the entity, since it is both required and unique. Designate the *sku* field as the primary key by annotating it with *javax.persistence.Id*. As long as the product SKU does not follow any specific pattern or convention, it can be automatically generated by a sequence. Annotate it with *javax.persistence.GeneratedValue* to have the value automatically generated and specify the strategy as *IDENTITY* to declare that the field is mapped to the primary key of the corresponding table and that the database will generate values for it:

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long sku
```

Named Query

The *featured* flag is used to designate some products to be showcased on the home page of the site. This leads to the requirement of a JPA query to find featured products. This query can be defined and declared in the entity as a named query:

```
@NamedQuery(name = "Product.findFeatured",
            query = "SELECT p FROM Product p WHERE p.featured = true")
```

Given an entity manager object, using a named query to find products is straightforward:

```
List<Product> products = em.createNamedQuery( "Product.findFeatured",
                                             Product.class ).getResultList();
```

Many to Many Relationship

Another mechanism to find and display products in the e-commerce application is to use a query to find products of a specific type. While various search and indexing solutions may be used to accomplish this application takes a more structured approach by classifying each product with a set of keywords that can be used to search and find them.

Model this by creating a keyword entity with a many to many relationship with product. Each product may be classified by multiple keywords, for example a given television is classified as both *TV* and *Electronics*. Similarly, the TV keyword refers to more than one television product in the database. Start by creating an entity with a single *keyword* field as its primary key:

```
@Entity
public class Keyword
{
    @Id
    private String keyword;
```

Generate getter and setter methods for this field.

For completeness, also create *equals*, *hashCode* and *toString* methods based on this field.

Declare a many to many relationship to the *Product* entity by declaring a list of Product objects as a field of the Keyword class and annotating it accordingly:

```
@ManyToMany(fetch = FetchType.EAGER, mappedBy = "keywords")
private List<Product> products;
```

Annotating a field with *javax.persistence.ManyToMany* tells JPA to use a join table to establish the relationship. The *mappedBy* element indicates that this class is not the **owning** side of this **bidirectional** relationship. The value of this element is the name of the field in the *Product* class that maps back to this class and declares the join table.

This relationship is bidirectional due to business and technical requirements. To find products classified with a given keyword, the Keyword entity is looked up and a getter is used to retrieve a list of products classified with that keyword. The Keyword entity must therefore be aware of the relationship. Conversely, it would be more natural to classify products from the Product side, than to look up multiple keyword objects and add the product to each of them. In fact, should the Product entity be unaware of the relationship, removing a product would involve searching for all keywords used to classify a product and updating them.

The relationship uses eager fetching. When a keyword is looked up, it is most often in response to a search and to return a list of products that have been classified with it.

The owning side of the relationship in the *Product* entity also declares a list of *Keyword* entities as a field and annotates this field:

```
@ManyToMany(fetch = FetchType.EAGER)
@JoinTable(name = "PRODUCT_KEYWORD",
    joinColumns = @JoinColumn(name = "SKU", referencedColumnName = "SKU"),
    inverseJoinColumns = @JoinColumn(name = "KEYWORD"),
    private List<Keyword> keywords;
```

The join table name is specified along with the two columns in the join table as well as the columns in the entity table that they point to.

There is no requirement to query the keywords of a product in the application, hence a getter method for the list of keywords is omitted. There is a need for a setter method in the *Product* class to allow a product to be classified. Conversely, the *Keyword* class only has a getter method and does not require a setter for its relationship.

In this application, there is no need to include the relationship field in any of the *equals*, *hashCode* or *toString* methods. In cases where the inclusion of such fields in these methods may be beneficial or even required, be careful to not include both sides of a bidirectional relationship as it can cause infinite loops between the two objects.

Case-insensitive search

While the actual keyword is the primary key of its entity, a JPA lookup may not be the optimal solution for finding products. That is because users often ignore the capitalization of a search query while computers in general and databases in particular find them significant. For this reason, the best solution is to do a case insensitive search to find keywords. Create a named query in the *Keyword* entity that uses the **Java Persistence Query Language (JPQL)** *UPPER()* function for this purpose:

```
@NamedQuery(name = "Keyword.findKeyword",
    query = "SELECT k FROM Keyword k WHERE UPPER(k.keyword) = UPPER(:query)")
```

5.3.4. Database setup

MySQL Database

Details of the database configuration remain outside the scope of this reference architecture, but as an example, some of the scripts used to set up the database used with the reference application and MySQL Database Server are provided in this document.

Create a new database for the product service:

```
CREATE DATABASE product;  
USE product;
```

Create a database user and grant this user the required privileges. In the earliest phases of development, this might be a user with full access to the databases, so that Hibernate can be used to create and drop tables while the design of the entities and their corresponding tables are being finalized:

```
CREATE USER 'product'@'localhost' IDENTIFIED BY 'password';  
GRANT USAGE ON . TO 'product'@'localhost' IDENTIFIED BY 'password';  
GRANT ALL PRIVILEGES ON product.* to product@localhost;
```

It is important to restrict user privileges in testing and staging environments, or even in the later stages of development, to avoid making incorrect assumptions based on a level of access that will not be granted to the application in a real production environment.

Create the product table to correspond to the entity. Once again, the syntax can be obtained from Hibernate when configured to generate the schema and log the database statements.

```
CREATE TABLE PRODUCT (SKU BIGINT NOT NULL AUTO_INCREMENT, DESCRIPTION VARCHAR(255),  
HEIGHT NUMERIC(5,2) NOT NULL, LENGTH NUMERIC(5,2) NOT NULL, NAME VARCHAR(255), HEIGHT  
NUMERIC(5,2) NOT NULL, WIDTH NUMERIC(5,2) NOT NULL, FEATURED BOOLEAN NOT NULL,  
AVAILABILITY INTEGER NOT NULL, IMAGE VARCHAR(255), PRICE NUMERIC(7,2) NOT NULL, PRIMARY  
KEY (SKU)) AUTO_INCREMENT = 10001;
```

Note that the primary key is set to automatically increment but to start at 10001, thereby insuring that the product SKU will always be at least 5 digits long.

The syntax for creating the keyword table is quite simple:

```
CREATE TABLE KEYWORD (KEYWORD VARCHAR(255) NOT NULL, PRIMARY KEY (KEYWORD));
```

Create a join table for the many to many relationship between product and keyword. This table has its own primary key, which has no business value and is not directly used in the application:

```
CREATE TABLE PRODUCT_KEYWORD (ID BIGINT NOT NULL AUTO_INCREMENT, KEYWORD VARCHAR(255) NOT  
NULL, SKU BIGINT NOT NULL, PRIMARY KEY (ID));
```

While JPA imposes restrictions to preserve the data integrity of your application, it can be useful to

also create database constraints to avoid incoherent data when it is loaded or modified through other means:

```
ALTER TABLE PRODUCT_KEYWORD ADD INDEX FK_PRODUCT_KEYWORD_PRODUCT (SKU), add constraint
FK_PRODUCT_KEYWORD_PRODUCT FOREIGN KEY (SKU) REFERENCES PRODUCT (SKU);
ALTER TABLE PRODUCT_KEYWORD ADD INDEX FK_PRODUCT_KEYWORD_KEYWORD (KEYWORD), add
constraint FK_PRODUCT_KEYWORD_KEYWORD FOREIGN KEY (KEYWORD) REFERENCES KEYWORD (KEYWORD);
```

This application pre-populates a number of products into the database:

```
INSERT INTO PRODUCT (DESCRIPTION, HEIGHT, LENGTH, NAME, WEIGHT, WIDTH, FEATURED,
AVAILABILITY, IMAGE, PRICE) VALUES ('HD LED Picture Quality<p/>ConnectShare Movie<p/>Wide
Color Enhancement<p/>Clear Motion Rate 60', 17.5, 29.1, 'ABC HD32CS5002 32-inch LED TV',
17, 3.7, true, 52, 'TV', 249.99 );
INSERT INTO PRODUCT (DESCRIPTION, HEIGHT, LENGTH, NAME, WEIGHT, WIDTH, FEATURED,
AVAILABILITY, IMAGE, PRICE) VALUES ('HD LED Picture Quality<p/>ConnectShare Movie<p/>Wide
Color Enhancement<p/>Clear Motion Rate 60', 22.3, 37.8, 'ABC HD42CS5002 42-inch LED TV',
20.9, 2.2, true, 64, 'TV', 424.95 );
INSERT INTO PRODUCT (DESCRIPTION, HEIGHT, LENGTH, NAME, WEIGHT, WIDTH, FEATURED,
AVAILABILITY, IMAGE, PRICE) VALUES ('Inverter Technology for even cooking<p/>Inverter
Turbo Defrost for quick defrosting<p/>9-Menu Category Sensor Cook system', 12, 22,
'Microtech MM-733N Microwave Oven, 1.6 Cubic Feet', 38.8, 19.5, true, 32, 'Microwave',
178 );
INSERT INTO PRODUCT (DESCRIPTION, HEIGHT, LENGTH, NAME, WEIGHT, WIDTH, FEATURED,
AVAILABILITY, IMAGE, PRICE) VALUES ('Intel Core i5-4210U 1.7 GHz (3 MB Cache)<p/>4 GB
DDR3L SDRAM<p/>0 GB 1 rpm 180 GB Solid-State Drive<p/>14-Inch Screen, Intel HD Graphics
4400<p/>Fedora 21 Operating System', 11.6, 20.4, 'HCM MegaBook 14-Inch Laptop', 6.2, 3.1,
true, 213, 'Laptop', 1095.99 );
INSERT INTO PRODUCT (DESCRIPTION, HEIGHT, LENGTH, NAME, WEIGHT, WIDTH, FEATURED,
AVAILABILITY, IMAGE, PRICE) VALUES ('Finished on all sides for versatile
placement<p/>Cinnamon Cherry finish<p/>Cinnamon Cherry', 19.5, 35.2, 'Coffee Table in
Cinnamon Cherry Finish', 26.9, 17.1, true, 23, 'CoffeeTable', 44.73 );
INSERT INTO PRODUCT (DESCRIPTION, HEIGHT, LENGTH, NAME, WEIGHT, WIDTH, FEATURED,
AVAILABILITY, IMAGE, PRICE) VALUES ('HD LED Picture Quality<p/>ConnectShare Movie<p/>Wide
Color Enhancement<p/>Clear Motion Rate 60', 33.5, 57.8, 'ABC HD65CS5002 65-inch LED TV',
72.5, 2.8, true, 76, 'TV', 999.00 );
INSERT INTO PRODUCT (DESCRIPTION, HEIGHT, LENGTH, NAME, WEIGHT, WIDTH, FEATURED,
AVAILABILITY, IMAGE, PRICE) VALUES ('Intel Core i5-4210U 1.7 GHz (3 MB Cache)<p/>4 GB
DDR3L SDRAM<p/>0 GB 1 rpm 180 GB Solid-State Drive<p/>15.6-Inch Screen, Intel HD Graphics
4400<p/>Fedora 21 Operating System', 11.9, 21.9, 'HCM MegaBook 15.6-Inch Laptop', 6.9, 3,
false, 251, 'Laptop', 1234.32 );
INSERT INTO PRODUCT (DESCRIPTION, HEIGHT, LENGTH, NAME, WEIGHT, WIDTH, FEATURED,
AVAILABILITY, IMAGE, PRICE) VALUES ('HD LED Picture Quality<p/>ConnectShare Movie<p/>Wide
Color Enhancement<p/>Clear Motion Rate 60', 24.7, 42.2, 'ABC HD47CS5002 47-inch LED TV',
28, 2.2, false, 76, 'TV', 529.00 );
INSERT INTO PRODUCT (DESCRIPTION, HEIGHT, LENGTH, NAME, WEIGHT, WIDTH, FEATURED,
```

```
AVAILABILITY, IMAGE, PRICE) VALUES ('HD LED Picture Quality<p/>ConnectShare Movie<p/>Wide
Color Enhancement<p/>Clear Motion Rate 60', 28.5, 48.9, 'ABC HD55CS5002 55-inch LED TV',
40.6, 2.2, false, 76, 'TV', 569.00 );
INSERT INTO PRODUCT (DESCRIPTION, HEIGHT, LENGTH, NAME, WEIGHT, WIDTH, FEATURED,
AVAILABILITY, IMAGE, PRICE) VALUES ('Inverter Technology for even cooking<p/>Inverter
Turbo Defrost for quick defrosting<p/>9-Menu Category Sensor Cook system', 14, 24,
'Microtech MM-733N Microwave Oven, 2.2 Cubic Feet', 45.6, 19.5, false, 41, 'Microwave',
135 );
INSERT INTO PRODUCT (DESCRIPTION, HEIGHT, LENGTH, NAME, WEIGHT, WIDTH, FEATURED,
AVAILABILITY, IMAGE, PRICE) VALUES ('Top lifts up and forward<p/>Hidden storage beneath
top<p/>Finished on all sides for versatile placement', 19.4, 41.1, 'Black Finish Coffee
Table', 67.6, 19, false, 6, 'CoffeeTable', 142.99 );
```

Six keywords are defined to classify these products:

```
INSERT INTO KEYWORD VALUES('Electronics');
INSERT INTO KEYWORD VALUES('Furniture');
INSERT INTO KEYWORD VALUES('TV');
INSERT INTO KEYWORD VALUES('Microwave');
INSERT INTO KEYWORD VALUES('Laptop');
INSERT INTO KEYWORD VALUES('Table');
```

These keywords are mapped to the sample products by inserting the required rows into the join table. In this example, use the product names to classify them without having a hard constraint on the generated SKU of the products:

```
INSERT INTO PRODUCT_KEYWORD (SKU, KEYWORD) SELECT SKU, 'Electronics' FROM PRODUCT WHERE
IMAGE IN ('TV', 'Microwave', 'Laptop');
INSERT INTO PRODUCT_KEYWORD (SKU, KEYWORD) SELECT SKU, 'Furniture' FROM PRODUCT WHERE
IMAGE = 'CoffeeTable';
INSERT INTO PRODUCT_KEYWORD (SKU, KEYWORD) SELECT SKU, 'Microwave' FROM PRODUCT WHERE
IMAGE = 'Microwave';
INSERT INTO PRODUCT_KEYWORD (SKU, KEYWORD) SELECT SKU, 'TV' FROM PRODUCT WHERE IMAGE =
'TV';
INSERT INTO PRODUCT_KEYWORD (SKU, KEYWORD) SELECT SKU, 'Laptop' FROM PRODUCT WHERE IMAGE
= 'Laptop';
INSERT INTO PRODUCT_KEYWORD (SKU, KEYWORD) SELECT SKU, 'Table' FROM PRODUCT WHERE IMAGE =
'CoffeeTable';
```

Datasource

Use the provided JBoss CLI script to configure a datasource for the *Product* and *Sales* server instances. As a first step, the downloaded MySQL JDBC driver JAR file has been installed in your JBoss EAP environment as a module. Custom JBoss EAP 7 modules are located directly under *jboss-eap-7.0/modules/*. The JDBC driver JAR file should be copied as part of the CLI configuration and reside under *jboss-eap-7.0/modules/com/mysql/main/*

Also note a module descriptor file called *module.xml* in this same directory:

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.1" name="com.mysql">
  <resources>
    <resource-root path="mysql-connector-java-5.1.34-bin.jar"/>
  </resources>

  <dependencies>
    <module name="javax.api"/>
    <module name="javax.transaction.api"/>
  </dependencies>
</module>
```

Notice that the module name is specified as *com.mysql*. The resources section specifies the driver archive as the only resource of this module while two other modules are listed as dependencies in the corresponding section. Verify the JBoss EAP server configuration and that a driver called *mysql* has been configured. Database drivers are configured in the datasources subsystem:

```
<subsystem xmlns="urn:jboss:domain:datasources:1.2">
  <datasources>
    ...
    <drivers>
      <driver name="mysql" module="com.mysql">
        <driver-class>com.mysql.jdbc.Driver</driver-class>
        <xa-datasource-class>com.mysql.jdbc.jdbc2.optional.MysqlXADataSource</xa-datasource-
class>
      </driver>
      <driver name="h2" module="com.h2database.h2">
        <xa-datasource-class>org.h2.jdbcx.JdbcDataSource</xa-datasource-class>
      </driver>
    </drivers>
  </datasources>
</subsystem>
```

Note that the driver definition refers to the module containing the database drivers.

The fully qualified class name of both the XA and non-XA drivers is used to refer to the driver classes.

This configuration is typically only required once per database and multiple datasources configured either on the server or within various applications can then take advantage of the same driver.

The CLI script also generates a datasource based on the specified property files.

```
<datasource jndi-name="java:jboss/datasources/ProductDS"
  pool-name="ProductDS" enabled="true" use-java-context="true">
  <connection-url>jdbc:mysql://product-db:3306/product</connection-url>
  <driver>mysql</driver>
  <security>
    <user-name>product</user-name>
    <password>password</password>
  </security>
</datasource>
```

Note that this datasource relies on the *mysql* JDBC driver.

5.4. RESTful API

5.4.1. Enabling JAX-RS support

To enable support for the **Java API for RESTful Web Services (JAX-RS)**, create a web descriptor for your application and provide a mapping for the standard JAX-RS servlet, which is `javax.ws.rs.core.Application`.

To create a `web.xml` descriptor, first navigate to the `src/main/webapp/WEB-INF` directory of your project in JBoss Developer Studio. Using either the *File* menu or by right-clicking on the folder, select to create a new artifact of type *Other*. Select *Web Descriptor* from the *JBoss Tools Web* category and press *Next*. Change the Servlet version to 3.1 and type the name as `web.xml`:

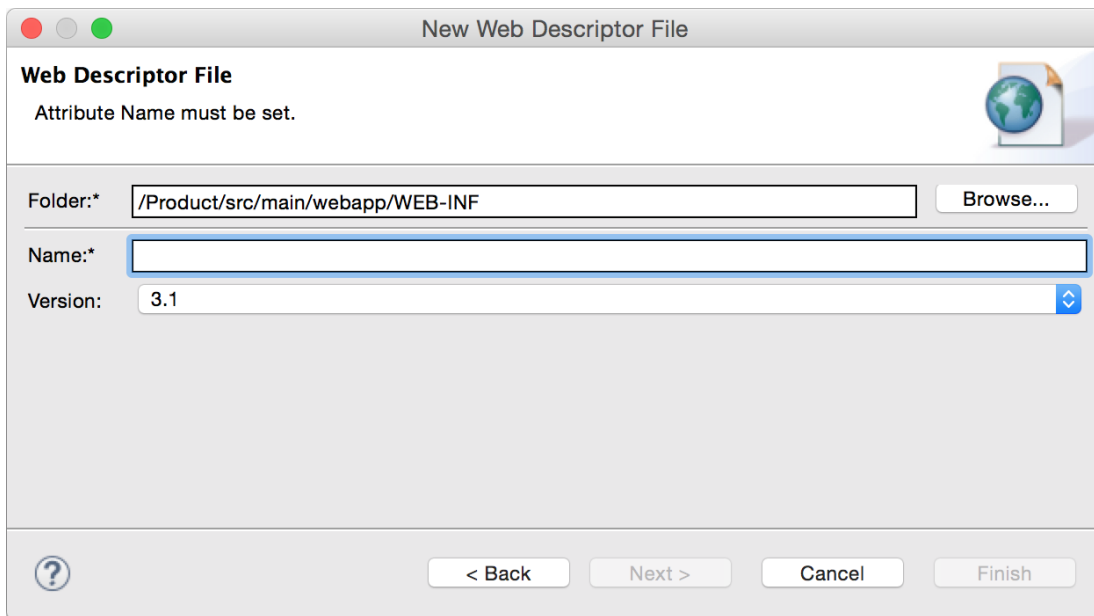


Figure 21. Create Web Descriptor

Click the *Finish* button to create the `web.xml` file. JBoss Developer Studio automatically opens this file in the web descriptor editor.

Right-click on the *web.xml* file name in the editor and select to create a new *Servlet Mapping*.

Enter the fully qualified class name of the JAX-RS Servlet and map it to the web application context root:

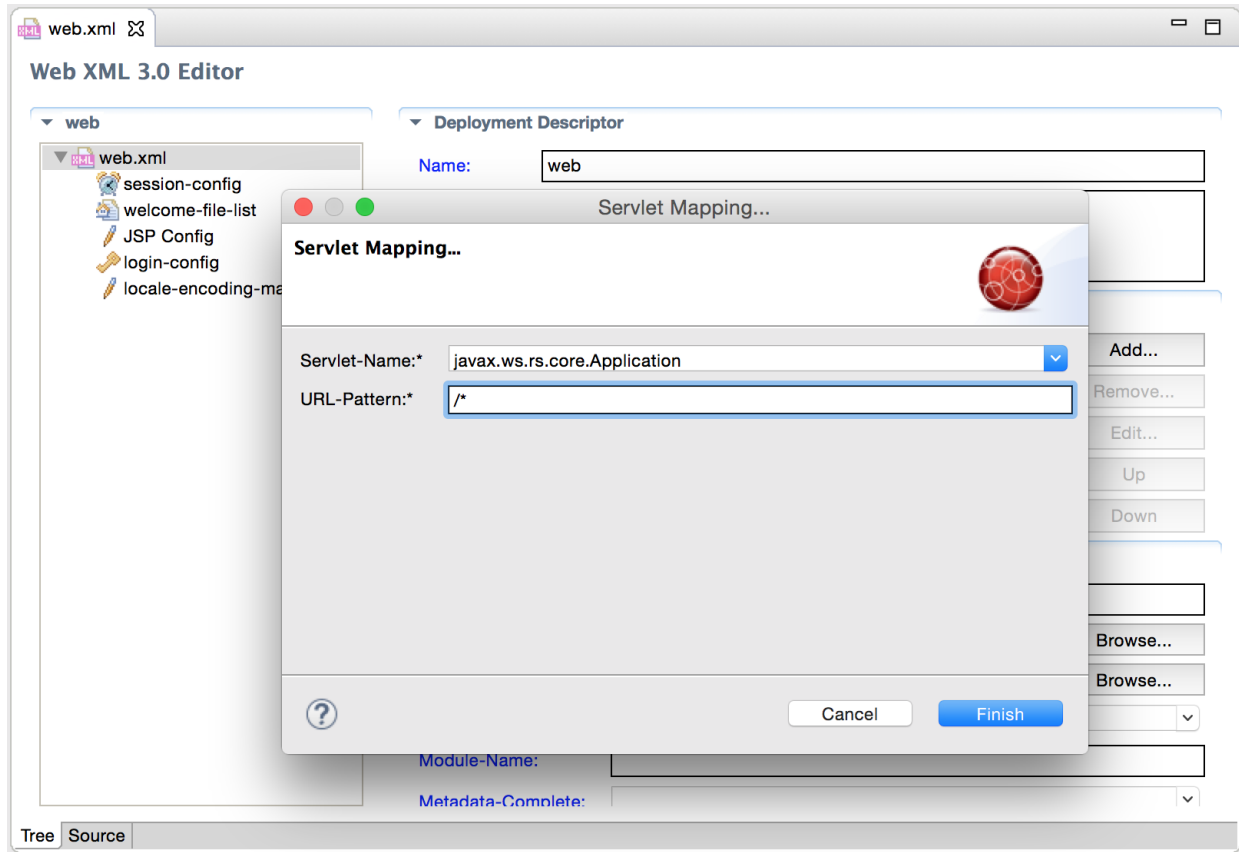


Figure 22. JAX-RS Servlet Mapping

The mapping specified for the JAX-RS servlet determines the root context of REST requests. The maven configuration of this web application has a project name of *product* and specifies war as its packaging format. As a result, this web application will have a root context of */product*. Using a wildcard URL pattern means that this will also be the top context of REST requests.

5.4.2. RESTful Service

To create a RESTful service, simply create a Java class and annotate the class with *javax.ws.rs.Path*.

Create a package called *com.redhat.refarch.microservices.product.service* in your project and place the *ProductService* class in this package.

This will be the only RESTful service in the product web application and it can therefore consume all requests targeted at the web application target. Set the service path to root:

```
package com.redhat.refarch.microservices.product.service;

import javax.ws.rs.Path;

@Path("/")
public class ProductService
{

}
```

To create an operation for this service, create a method and annotate it with a *Path*. For example, create a method called *addProduct* that both takes and returns a product and specify its relative context as */products*.

Note that the URL of this operation will be a combination of the path to the server, the web application, the JAX-RS servlet, the service and the operation itself. Assuming a local development server listening on <http://localhost:8080> and the current maven build file, which generates a web application called *product.war*, along with the root relative context given to the JAX-RS servlet and product service itself, the path pieces are <http://localhost:8080> and */product* and */* and */products*.

The final path to this operation is: <http://localhost:8080/product/products>

Also use annotations to declare the HTTP method supported by this operation as well as the media type consumed and produced.

Create this operation with the assumption that the product will be posted to the service in JSON form and also returned in *JSON* form:

```
@Path("/products")
@POST
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Product addProduct(Product product)
{
    return product;
}
```

Build the project by running the install goal of maven, either through JBDS by choosing *Run As Maven* build or in command line: `mvn install`

Once built, a *product.war* archive file will be generated in the *target* directory. Deploy this web application archive to JBoss EAP.

Once deployed, test invoking the add product operation. Use a REST client of your choice. For example, if using the **cURL** command line tool:

```
# curl -X POST -H 'Content-Type: application/json'
-H 'Accept: application/json'
-d @request.json
http://localhost:8080/product/products

{"sku":null,"name":"Product","description":"The product
description","length":40,"width":40,"height":40,"weight":1040,"featured":true,"availabili
ty":10,"price":19.95,"image":null}
```

In the above example, the request.json file would be created with the following content:

```
{
  "name": "Product",
  "description": "The product description",
  "length": 40,
  "width": 40,
  "height": 40,
  "weight": 1040,
  "availability": 10,
  "price": 19.95,
  "featured": true
}
```

Notice that the JAX-RS implementation automatically deserializes the JSON request to the provided Java type. The missing fields are left as null and this can be observed in the response.

Deserializing XML to Java types requires an extra step. JBoss EAP 7 can use **JAXB** to convert Java to XML and back, but Java classes must be annotated as XML types. Modify the *Product* class and annotate it as a JAXB root element:

```
@Entity
@NamedQuery(name = "Product.findFeatured",
            query = "SELECT p FROM Product p WHERE p.featured = true")
@XmlRootElement
public class Product
{
```

Modify the add product annotation to include XML as an acceptable media type for both the request and response:

```
@Path("/products")
@POST
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Product addProduct(Product product)
```

Note the use of curly braces to specify a list of options instead of a single string value.

Redeploy the web application and issue a similar REST request in XML, specifying both the request and response types as *application/xml*: Once again, using the **cURL** command line tool:

```
# curl -X POST -H 'Content-Type: application/xml'
-H 'Accept: application/xml'
-d @request.xml
http://localhost:8080/product/products
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><product><description>The product
description</description><height>40</height><length>40</length><name>Product</name><weigh
t>1040</weight><width>40</width></product>
```

Note that while serializing the response, JAXB simply omits elements with a null value. The request XML looks as follows:

```
<product>
  <description>The product description</description>
  <height>40</height>
  <length>40</length>
  <name>Product</name>
  <weight>1040</weight>
  <width>40</width>
  <price>40</price>
  <availability>40</availability>
</product>
```

5.4.3. Transactional Behavior

Some operations such as JPA persistence require a transactional context to proceed. The **Narayana 5** community project, used as the basis of the transactions subsystem in JBoss EAP 7, provides REST transactions. However JBoss EAP 7 does not support this feature and only includes support for *WS-BA* and *WS-AT*. This reference application simply relies on transactions with the scope of individual REST services. To create a transactional context for a REST operation, you can use either implicit container managed transactions or user transactions.

To leverage user transactions, declare the required dependencies on the transaction API and inject a user transaction object. Once this is set up, you can start a transaction at any point by calling the *begin()* method. The entity manager can enlist the database driver in the transaction through its *joinTransaction()* method. After the success or failure of the operation, either the *commit()* or the *rollback()* methods can be called to respectively commit or abandon the transaction.

It is often much simpler to allow the application server to manage transactions. This functionality is available for EJB3 session beans. Any class annotated as a REST service can also be enhanced to act as a stateless session bean. Annotate the *ProductService* class as a stateless session bean with no interface (therefore only a local bean):

```
@Path("/")
@Stateless
@LocalBean
public class ProductService
{
```

Inject an entity manager to find, query, save, update and delete JPA entities:

```
@PersistenceContext
private EntityManager em;
```

Modify the add product operation to use this entity manager and persist the supplied product:

```
@Path("/products")
@POST
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Product addProduct(Product product)
{
    em.persist( product );
    return product;
}
```

Redeploy the application and test it again by invoking the add product operation. For example, using cURL:

```
# curl -X POST -H 'Content-Type: application/json'
-H 'Accept: application/json'
-d @request.json
http://localhost:8080/product/products

{"sku":10012,"name":"Product","description":"The product
description","length":40,"width":40,"height":40,"weight":1040,"featured":true,"availabili
ty":10,"price":19.95,"image":null}
```

Notice that this time, the SKU has been filled in and returned as 10012. That is because the object was persisted to the database and its primary key was auto generated. The table was created with SKU as a primary key that is auto-incremented and starts with 10001:

```
...PRIMARY KEY (SKU)) AUTO_INCREMENT = 10001;
```

After inserting 11 sample products into the database using SQL scripts, the next inserted row should indeed be assigned a primary key of 10012.

5.4.4. Logging

Use the Java Logging API to plug into the JBoss EAP logging mechanism. Create a logger field in your service class and use the class name as the name:

```
@PersistenceContext
private EntityManager em;

private Logger logger = Logger.getLogger( getClass().getName() );
```

For convenience, create *logInfo* and *logError* methods that log statements with the respective verbosity levels:

```
private void logInfo(String message)
{
    logger.log( Level.INFO, message );
}

private void logError(String message)
{
    logger.log( Level.SEVERE, message );
}
```

Use these convenience methods to generate log statements that can help troubleshoot and verify application functionality:

```
public Product addProduct(Product product)
{
    logInfo( "Will persist product " + product );
    em.persist( product );
    return product;
}
```

5.4.5. Error handling

Well-designed RESTful services use standard HTTP codes along with descriptive information to communicate errors. The JAX-RS specification allows developers to return HTTP errors by throwing *javax.ws.rs.WebApplicationException*.

To validate a request and return an error when invalid, use HTTP code 422:

```
throw new WebApplicationException( 422 );
```

While this simple approach succeeds in communicating a correct and meaningful HTTP error code back to the client, it fails to provide any details or even return the response in the expected format, which is typically either JSON or XML. To return the proper descriptive response with the error, create a `javax.ws.rs.core.Response` object and pass it to the constructor of `javax.ws.rs.WebApplicationException`.

Create an *Error* class to use to represent various potential errors.

Like other object types returned by the RESTful service, annotate this class as `javax.xml.bind.annotation.XmlRootElement` so that it can be serialized to both JSON and XML. Create the following fields in the Error class:

- `int code`: The HTTP error code that is returned
- `String message`: Descriptive message that explain the cause of the error
- `String details`: Further details about the error, for example the exception stack

The Error class would start as follows:

```
package com.redhat.refarch.microservices.product.model;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class Error
{
    private int code;
    private String message;
    private String details;
```

Provide convenient constructors to instantiate the class:

```
public Error(int code, String message, Throwable throwable)
{
    this.code = code;
    if( message != null )
    {
        this.message = message;
    }
    else if( throwable != null )
    {
        this.message = throwable.getMessage();
    }
    if( throwable != null )
    {
        StringWriter writer = new StringWriter();
        throwable.printStackTrace( new PrintWriter( writer ) );
        this.details = writer.toString();
    }
}

public Error(int code, String message)
{
    this( code, message, null );
}

public Error(int code, Throwable throwable)
{
    this( code, null, throwable );
}
```

There is no legitimate use case for a default constructor with no arguments, however JAXB requires such a constructor. Providing a private constructor allows you to satisfy this JAXB requirement without promoting the incorrect use of the class:

```
@SuppressWarnings("unused")
private Error()
{
}
```

Remember that serialization to either JSON or XML is based on JavaBean properties. Provide getter and setter methods for only the second and third field, to include them in the response:

```
public String getMessage()
{
    return message;
}

public void setMessage(String message)
{
    this.message = message;
}

public String getDetails()
{
    return details;
}

public void setDetails(String details)
{
    this.details = details;
}
```

Also provide a convenience method to return a *WebApplicationException* based on this Error response:

```
public WebApplicationException asException()
{
    ResponseBuilder responseBuilder = Response.status( code );
    responseBuilder = responseBuilder.entity( this );
    return new WebApplicationException( responseBuilder.build() );
}
```

This class may now be used to return meaningful error messages. For example, try validating that the product being added has its name, price and availability set:

```
@Path("/products")
@POST
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Product addProduct(Product product)
{
    if( product.getAvailability() == null ||
        product.getName() == null || product.getPrice() == null )
    {
        throw new Error( 422, "Validation Error" ).asException();
    }
}
```

The *Error* class can also map any potential Java exceptions into a response code and description that can be consumed by a RESTful client. For example, an attempt to store a product that violates database constraints causes the entity manager to throw a *javax.persistence.PersistenceException*, which is a runtime exception:

```
try
{
    logInfo( "Will persist product " + product );
    em.persist( product );
    return product;
}
catch( RuntimeException e )
{
    logError( "Got exception " + e.getMessage() );
    throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR,
                    e ).asException();
}
```

Redeploy the application and test error handling. Try to add a product using JSON without including the price, with a *request.json* file as follows:

```
{
  "name": "Product",
  "description": "The product description",
  "length": 40,
  "width": 40,
  "height": 40,
  "weight": 1040,
  "availability": 10,
  "featured": true
}
```

Use the *-i* parameter with cURL to include the response headers:

```
# curl -X POST -H 'Content-Type: application/json'
               -H 'Accept: application/json'
               -d @request.json
               -i
               http://localhost:8080/product/products
```

```
HTTP/1.1 422 Unprocessable Entity
Server: Apache-Coyote/1.1
Content-Type: application/json
Transfer-Encoding: chunked
Date: ...
```

```
{"message": "Validation Error", "details": null}
```

Notice that the error code 422 is included in the response along with its default HTTP description. Now, try the corresponding XML request:

```
<product>
  <description>The product description</description>
  <height>40</height>
  <length>40</length>
  <name>Product</name>
  <weight>1040</weight>
  <width>40</width>
  <price>40</price>
</product>
```

The response looks a bit different this time:

```
# curl -X POST -H 'Content-Type: application/xml'
               -H 'Accept: application/xml'
               -d @request.xml
               -i
               http://localhost:8080/product/products

HTTP/1.1 422 Unprocessable Entity
Server: Apache-Coyote/1.1
Content-Type: application/xml
Content-Length: 105
Date: Sat, 28 Mar 2015 00:28:48 GMT

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<error><message>Validation Error</message></error>
```

Most notably, the JSON response includes a null detail whereas the XML response omits it entirely. These differences may be attributed to both the protocol as well as the serialization libraries.

Include the price to pass validation but change one of the other fields to an invalid value that will fail persistence. For example, set the weight to 9999999 which violates database constraint for the corresponding column. The details of the response will include the full stack of the exception:

HTTP/1.1 500 Internal Server Error

Server: Apache-Coyote/1.1

Content-Type: application/xml

Transfer-Encoding: chunked

Date: ...

Connection: close

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

```
<error>
```

```
  <details>javax.persistence.PersistenceException: org.hibernate.exception.DataException: could not execute statement
```

```
    at org.hibernate.ejb.AbstractEntityManagerImpl.convert(AbstractEntityManagerImpl.java:1387)
```

```
    at org.hibernate.ejb.AbstractEntityManagerImpl.convert(AbstractEntityManagerImpl.java:1310)
```

```
    at org.hibernate.ejb.AbstractEntityManagerImpl.convert(AbstractEntityManagerImpl.java:1316)
```

```
    at org.hibernate.ejb.AbstractEntityManagerImpl.persist(AbstractEntityManagerImpl.java:881)
```

```
    at org.jboss.as.jpa.container.AbstractEntityManager.persist(AbstractEntityManager.java:563)
```

```
    at
```

```
com.redhat.refarch.microservices.product.service.ProductService.addProduct(ProductService.java:43)
```

```
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
```

```
    at
```

```
    ... xxx more
```

```
Caused by: com.mysql.jdbc.MysqlDataTruncation: Data truncation: Out of range value for column 'WEIGHT' at row 1
```

```
    at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:3885)
```

```
    at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:3823)
```

```
    at com.mysql.jdbc.MysqlIO.sendCommand(MysqlIO.java:2435)
```

```
    at com.mysql.jdbc.MysqlIO.sqlQueryDirect(MysqlIO.java:2582)
```

```
    at com.mysql.jdbc.ConnectionImpl.execSQL(ConnectionImpl.java:2530)
```

```
    at com.mysql.jdbc.PreparedStatement.executeInternal(PreparedStatement.java:1907)
```

```
    at com.mysql.jdbc.PreparedStatement.executeUpdate(PreparedStatement.java:2141)
```

```
    at com.mysql.jdbc.PreparedStatement.executeUpdate(PreparedStatement.java:2077)
```

```
    at com.mysql.jdbc.PreparedStatement.executeUpdate(PreparedStatement.java:2062)
```

```
    at
```

```
org.jboss.jca.adapters.jdbc.WrappedPreparedStatement.executeUpdate(WrappedPreparedStatement.java:493)
```

```
    at
```

```
org.hibernate.engine.jdbc.internal.ResultSetReturnImpl.executeUpdate(ResultSetReturnImpl.java:186)
```

```
    ... 106 more
```

```
</details>
```

```
  <message>org.hibernate.exception.DataException: could not execute statement</message>
```

```
</error>
```

Note that simply catching and mapping Java exceptions to a service error risks exposing any content contained in the exception stack. Employ caution to not inadvertently expose vulnerabilities or any other information that you do not wish to expose as part of the exception stack trace.

5.4.6. Resource API design

While no strict standards govern RESTful service API design, conventions and common practice often go a long way in promoting consistent behavior. Familiar API design helps increase productivity and reduce misunderstanding and developer error.

This document proposes a set of URL patterns that are combined with standard HTTP methods to provide full create, read, update and delete (CRUD) capability for a resource using RESTful API.

There are some disagreements on the details of this approach but a large number of systems use some slight variation of this.

Relative context

Use the plural form of the resource name as the relative URL of each CRUD operation. For operations involving products, use */products* as the path or the first part of the path.

For example:

```
@Path("/products")
@POST
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Product addProduct(Product product)
```

Create

Use HTTP POST to add a new resource instance. Specify the path to this operation as the plural form of the resource name and receive the resource as the request content.

Return the persisted resource, which reflects any potential changes made to the entity upon saving it, including any automatically generated or sequence identifier.

The previously created add product method is an example of this:

```
@Path("/products")
@POST
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Product addProduct(Product product)
{
    try
    {
        logInfo( "Will persist product " + product );
        em.persist( product );
        return product;
    }
    catch( RuntimeException e )
    {
        logError( "Got exception " + e.getMessage() );
        throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR,
                        e ).asException();
    }
}
```

Read

Search

Use HTTP GET with the same path (plural form of the resource name) to retrieve resources. Query parameters may be used to filter the search results based on provided criteria:

```
@GET
@Path("/products")
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Collection<Product> getProducts(@Context UriInfo uriInfo)
{
    try
    {
        MultivaluedMap<String, String> queryParams =
            uriInfo.getQueryParameters();
    }
}
```

The product service will look for the featured query parameter and if present, returns only the products that are flagged as featured:

```
if( queryParams.containsKey( "featured" ) )
{
    return em.createNamedQuery( "Product.findFeatured",
                               Product.class ).getResultList();
}
```

In the absence of the *featured* flag, if one or more keywords are provided in the query, products classified with these keywords will be returned. Use the named query in the *Keyword* entity to perform a case-insensitive search for those keywords. If a keyword is not found, there are no associated products to return. It is also possible that a keyword is declared but no product has yet been classified with it:

```
else if( queryParams.containsKey( "keyword" ) )
{
    Collection<Product> products = new HashSet<Product>();
    for( String keyword : queryParams.get( "keyword" ) )
    {
        try
        {
            TypedQuery<Keyword> query =
            em.createNamedQuery( "Keyword.findKeyword", Keyword.class );
            query.setParameter( "query", keyword );
            Keyword keywordEntity = query.getSingleResult();
            List<Product> keywordProducts = keywordEntity.getProducts();
            logInfo( "Found " + keyword + ": " + keywordProducts );
            products.addAll( keywordProducts );
        }
        catch( NoResultException e )
        { //keyword not found, which is acceptable
        }
    }
    return products;
}
```

When searching for products by keyword, it is conceivable and sometimes even likely that a single product will match two or more specified keywords. To avoid providing duplicates in the results, use a *HashSet* class as the *Collection* implementation. The *HashSet* class ensures unique contents by using the *equals* and *hashCode* methods of the provided type. In this case, both *equals* and *hashCode* methods have been implemented for the *Product* class.

This method uses the JPA relationship between keywords and products to find all products classified with each keyword, aggregating and returning a unique set of results.

If neither the featured nor keyword query parameters are provided, the client is presumably requesting a list of all products. This may be an acceptable use case for some services, but the product service does not support this feature. Return a descriptive error message with a 400 error code:

```
else
{
    throw new Error( HttpURLConnection.HTTP_BAD_REQUEST,
        "All products cannot be returned" ).asException();
}
}
catch( RuntimeException e )
{
    throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR, e ).asException();
}
```

Once again, any unexpected error is mapped to an error response with HTTP error code 500.

Lookup

Another type of read operation is the direct lookup by a resource by its unique identifier. The convention for retrieving a known resource is to issue a GET request to an address that includes the relative URL of the resource type and is followed by the unique resource identifier. In the case of products, this would be */products/sku*.

Use the *javax.ws.rs.PathParam* annotation to use and specify a variable in the path of the operation:

```
@GET
@Path("/products/{sku}")
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Product getProduct(@PathParam("sku") Long sku)
{
    logInfo( "SKU is " + sku );
    Product product = em.find(Product.class, sku );
    if( product == null )
    {
        throw new Error( HttpURLConnection.HTTP_NOT_FOUND,
            "Product not found" ).asException();
    }
    return product;
}
```

If a product with the specified SKU is not found, an HTTP error code 404 is returned along with a descriptive message in the accepted media type.

Note that the product SKU is expected to conform to the Java *long* format. The REST framework is tasked with validating and converting the provided characters to the specified type. While convenient, the disadvantage of this approach is that any validation error is handled by the server. Providing a non-numeric value to the JAX-RS implementation provided with JBoss EAP 7 returns a 400 HTTP error code, indicating a *Bad Request*. An error message in the accepted media type cannot be returned alongside the error code.

Update

When updating a resource, the request does not always include every resource attribute. It is critical to distinguish between the intent to only update the provided attributes without modifying the others, versus removing all other attributes that were not included in the update request. These two intents can be considered separate operations, where one is a full update and the other a partial one.

While there is less agreement and consistency in the conventions used to distinguish between a full and partial update in a RESTful API, one common approach is to use the distinct HTTP methods of *PUT* and *PATCH*, with the latter indicating a partial update.

What follows is an example of a simplistic and incomplete implementation of partial update with the *PATCH* method. The advantage of this approach is that the implementation effort is minimal and we rely on the existing framework to provide the operation and map the request and response. The notable **limitation** is that any null value provided in a partial update call is ignored and thus the operation cannot be used to remove a value. In other words, when entirely relying on the existing JAX-RS framework, it is not possible to distinguish between a field value that is not provided and one that is being set to null. To provide this capability, the code must take further control over the source request by either manually parsing the request media type, or implementing an intercepter to assist in achieving the objective.

Create a utility class that uses the JavaBeans API to copy fields from one object to the other. Start by using a map to cache the bean property descriptors:

```
package com.redhat.refarch.microservices.utils;

...
public class Utils
{

    private static final Map<Class<?>, PropertyDescriptor[]>
        beanDescriptors = new HashMap<Class<?>, PropertyDescriptor[]>();
```

Create a method to introspect bean classes and return the bean property descriptors, while using the cache:

```
private static PropertyDescriptor[] getBeanDescriptors(Class<?> clazz)
{
    PropertyDescriptor[] descriptors = beanDescriptors.get( clazz );
    if( descriptors == null )
    {
        try
        {
            BeanInfo beanInfo = Introspector.getBeanInfo( clazz );
            descriptors = beanInfo.getPropertyDescriptors();
            beanDescriptors.put( clazz, descriptors );
        }
        catch( IntrospectionException e )
        {
            throw new IllegalStateException( e );
        }
    }
    return descriptors;
}
```

Create a generic method that copies the properties of one bean to another and include a flag in the method signature to determine the behavior when a source bean property is *null*.

This flag provides the option to skip copying over *null* values, thereby leaving such destination properties unchanged, or setting them to *null*:

```
public static <T> void copy(T source, T destination, boolean skipIfNull)
{
   PropertyDescriptor[] descriptors = getBeanDescriptors(
        source.getClass() );
    for( PropertyDescriptor descriptor : descriptors )
    {
        try
        {
            if( "class".equals( descriptor.getName() ) )
            {
                //Class is not a regular JavaBeans property!
                continue;
            }
            Method readMethod = descriptor.getReadMethod();
            Method writeMethod = descriptor.getWriteMethod();
            if( readMethod == null || writeMethod == null )
            {
                //Property must be read/write to copy
                continue;
            }
            Object value = readMethod.invoke( source );
            if( value == null && skipIfNull == true )
            {
                //As per the flag, do not copy null properties
                continue;
            }
            else
            {
                writeMethod.invoke( destination, value );
            }
        }
        catch( ReflectiveOperationException e )
        {
            throw new IllegalStateException( e );
        }
    }
}
```

Full update

Create a method that listens for PUT requests on an address that includes the relative URL of the resource type and is followed by the unique resource identifier. In the case of products, this would be */products/sku*.

This method uses the product SKU to retrieve the product and update all its fields with the provided product:

```
@PUT
@Path("/products/{sku}")
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Product updateProduct(@PathParam("sku") Long sku, Product product)
{
    Product entity = getProduct( sku );
    try
    {
        //Ignore any attempt to update product SKU:
        product.setSku( sku );
        Utils.copy( product, entity, false );
        em.merge( entity );
        return product;
    }
    catch( RuntimeException e )
    {
        throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR,
                        e ).asException();
    }
}
```

This method uses the existing product lookup operation to load the JPA entity. Notice that this calls takes place outside the try block so that it is not caught and wrapped again, but instead directly bubbles up and results in the original HTTP error code and description.

It then calls the previously written utility method to update it. Providing *false* as the third argument causes the method to overwrite all of the entity properties with the provided values, even if some of the provided values are null.

The *merge* method of the entity manager updates the database.

Partial update

The REST API for partial update is similar to Full update, but instead uses the PATCH method. JAX-RS does not provide native support for HTTP PATCH so you have to first declare an annotation for this purpose:

```
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@HttpMethod("PATCH")
public @interface PATCH
{
}
```

Create a method identical to the one used for full update, with the different HTTP method and a flag to ask the utility method to ignore null values:

```
@PATCH
@Path("/products/{sku}")
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Product partiallyUpdateProduct(@PathParam("sku") Long sku,
                                     Product product)
{
    Product entity = getProduct( sku );
    try
    {
        //Ignore any attempt to update product SKU:
        product.setSku( sku );
        Utils.copy( product, entity, true );
        em.merge( entity );
        return product;
    }
    catch( RuntimeException e )
    {
        throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR,
                        e ).asException();
    }
}
```

Delete

The REST API to delete a resource uses the same address of the resource type followed by the unique identifier of the resource entity:

```
@DELETE
@Path("/products/{sku}")
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public void deleteProduct(@PathParam("sku") Long sku)
{
    Product product = getProduct( sku );
    try
    {
        em.remove( product );
    }
    catch( RuntimeException e )
    {
        throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR,
                        e ).asException();
    }
}
```

5.4.7. Other RESTful operations

Not all product requirements always neatly fit into the resource model. Model other operations in a consistent and similar way. Create a method to add a keyword to the database that can later be used to classify product.

This can be modeled after a Create resource operation:

```
@Path("/keywords")
@POST
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Keyword addKeyword(Keyword keyword)
{
    try
    {
        logInfo( "Will persist keyword " + keyword );
        em.persist( keyword );
        return keyword;
    }
    catch( RuntimeException e )
    {
        logError( "Got exception " + e.getMessage() );
        throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR,
                        e ).asException();
    }
}
```

Once keywords have been added using the above operation, they can be used to classify a product:

```
@POST
@Path("/classify/{sku}")
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public void classifyProduct(@PathParam("sku") Long sku,
                           List<Keyword> keywords)
{
    Product product = getProduct( sku );
    logInfo( "Asked to classify " + product + " as " + keywords );
    try
    {
        product.setKeywords( keywords );
        em.merge( product );
    }
    catch( RuntimeException e )
    {
        throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR,
                        e ).asException();
    }
}
```

Once again, keep calls to other methods outside the try loop so the exceptions are not caught and wrapped inside an internal server error.

To adjust product availability in response to a purchase, the service client is expected to match each product SKU with the quantity that's been ordered. Create a java bean to represent the inventory adjustment:

```
package com.redhat.refarch.microservices.product.model;

public class Inventory
{
    private long sku;
    private int quantity;

    public long getSku()
    {
        return sku;
    }

    public void setSku(long sku)
    {
        this.sku = sku;
    }

    public int getQuantity()
    {
        return quantity;
    }

    public void setQuantity(int quantity)
    {
        this.quantity = quantity;
    }

    @Override
    public String toString()
    {
        return "Inventory [sku=" + sku + ", quantity=" + quantity + "];"
    }
}
```

Create an operation to reduce product availability based on the order quantity:

```
@POST
@Path("/reduce/")
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public void reduceInventory(Inventory[] inventoryAdjustment)
{
    ...
}
```

5.4.8. Pessimistic Locking

Certain concurrency issues would benefit from pessimistic locking to avoid data modification while a transaction is inflight.

Some distributed databases cannot support pessimistic locking but where the database provides the required support, JPA exposes this capability through its API.

When fulfilling orders, product availability is checked against the order quantity. With the potential for concurrent requests, it is crucial to ensure that the product availability is not prone to changes until the transaction is completed and the availability is updated.

Specify the lock type while looking up the product:

```
Product product = em.find( Product.class, inventory.getSku(),
                           LockModeType.PESSIMISTIC_WRITE );
```

This results in a *SELECT FOR UPDATE* database query that locks the affected rows until the transaction is either committed or rolled back.

Compare the amount of inventory adjustment with the product availability and if the requested quantity exceeds available inventory, return an HTTP error code 409 with a descriptive message that states there was insufficient availability for the given product SKU:

```
try
{
    logInfo( "Asked to reduce inventory: " +
            Arrays.toString( inventoryAdjustment ) );
    for( Inventory inventory : inventoryAdjustment )
    {
        Product product = em.find( Product.class, inventory.getSku(),
                                   LockModeType.PESSIMISTIC_WRITE );
        logInfo( "Looked up product as " + product );
        if( product == null )
        {
            throw new Error( HttpURLConnection.HTTP_NOT_FOUND,
                            "Product not found" ).asException();
        }
        int availability = product.getAvailability();
        if( inventory.getQuantity() > availability )
        {
            String message = "Insufficient availability for "
                            + inventory.getSku();
            throw new Error( HttpURLConnection.HTTP_CONFLICT,
                            message ).asException();
        }
        else
        {
            product.setAvailability(
                availability - inventory.getQuantity() );
            em.merge( product );
            logInfo( "Saved " + product );
        }
    }
}
catch( WebApplicationException e )
{
    throw e;
}
catch( RuntimeException e )
{
    throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR, e ).asException();
}
```

5.4.9. Sales service

Create a second JBDS project called Sales to develop a second service that handles customer data maintenance and order management.

Follow the same steps that were used to create the product project, starting by Creating a Maven Project. Optionally, you can also duplicate the existing product project in JBoss Developer Studio, renaming it to sales and making the necessary changes.

Use a datasource file called *sales-ds.xml* with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources xmlns="http://www.jboss.org/ironjacamar/schema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.jboss.org/ironjacamar/schema
http://docs.jboss.org/ironjacamar/schema/datasources_1_0.xsd">
  <datasource jndi-name="java:jboss/datasources/SalesDS"
    pool-name="SalesDS" enabled="true" use-java-context="true">
    <connection-url>jdbc:mysql://sales-db:3306/sales</connection-url>
    <driver>mysql</driver>
    <security>
      <user-name>sales</user-name>
      <password>password</password>
    </security>
  </datasource>
</datasources>
```

Reference the new and correct datasource in the *persistence.xml* configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
  xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence\_2\_1.xsd">
  <persistence-unit name="primary">
    <jta-data-source>java:jboss/datasources/SalesDS</jta-data-source>
    <properties>
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.MySQLDialect" />
    </properties>
  </persistence-unit>
</persistence>
```

Follow the same instructions provided in the Persistence Entity section to create a JPA bean to represent a customer. Include JAXB annotations to allow XML serialization for this bean.

Also include a named query to find customers by their username, as they attempt to log in to the application:

```
package com.redhat.refarch.microservices.sales.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.NamedQuery;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
@Entity
@NamedQuery(name = "Customer.findByUsername",
    query = "SELECT c FROM Customer c WHERE c.username = :username")
public class Customer
{

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String address;
```

```
private String telephone;
private String email;
private String username;
private String password;

public Long getId()
{
    return id;
}

public void setId(Long id)
{
    this.id = id;
}

public String getName()
{
    return name;
}

public void setName(String name)
{
    this.name = name;
}

public String getAddress()
{
    return address;
}

public void setAddress(String address)
{
    this.address = address;
}

public String getTelephone()
{
    return telephone;
}

public void setTelephone(String telephone)
{
    this.telephone = telephone;
}

public String getEmail()
{
    return email;
}
```

```
}

public void setEmail(String email)
{
    this.email = email;
}

public String getUsername()
{
    return username;
}

public void setUsername(String username)
{
    this.username = username;
}

public String getPassword()
{
    return password;
}

public void setPassword(String password)
{
    this.password = password;
}

@Override
public String toString()
{
    return "Customer [id=" + id + ", name=" + name +
        ", address=" + address + ", telephone=" + telephone + ", email=" + email + ",
username=" + username + ", password=" + password + "];"
}
}
```

Model orders as logically dependent on customers. In other words, an order can only be created by a customer and exists only as part of the customer data. Similarly, order items are parts of an order that exist within an order.

Use a Java enumeration to define the status of an order. Create two named queries for order, allowing the service to find all orders for a customer or to find customer orders that are in a given status.

Create one to many mapping between customer and order, as well as between order and order item. When creating getters for bean properties, remember to omit those fields that you do not want returned as part of the entity. Instead, use a different method name to allow retrieval of the field or relationship by the service while excluding it from the default serialization behavior. For example, use *retrieveCustomer* instead of *getCustomer*:

```
package com.redhat.refarch.microservices.sales.model;

import java.util.Date;
import java.util.List;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.OneToMany;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
@Entity(name = "Orders")
@NamedQueries({@NamedQuery(name = "Order.findByCustomer", query = "SELECT o FROM Orders o WHERE o.customer = :customer"), @NamedQuery(name = "Order.findByOrderStatus", query = "SELECT o FROM Orders o WHERE o.customer = :customer AND o.status = :status")})
public class Order
{

    public enum Status
    {
        Initial, InProgress, Canceled, Paid, Shipped, Completed
    }

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private Status status;
    private Long transactionNumber;
    private Date transactionDate;

    @ManyToOne(optional = false, fetch = FetchType.EAGER)
    @JoinColumn(name = "CUSTOMER_ID", referencedColumnName = "ID")
```

```
private Customer customer;

@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER, mappedBy = "order")
private List<OrderItem> orderItems;

public Long getId()
{
    return id;
}

public void setId(Long id)
{
    this.id = id;
}

public Status getStatus()
{
    return status;
}

public void setStatus(Status status)
{
    this.status = status;
}

//Avoid getter so it is not included in automatic serialization
public Customer retrieveCustomer()
{
    return customer;
}

public void setCustomer(Customer customer)
{
    this.customer = customer;
}

public List<OrderItem> getOrderItems()
{
    return orderItems;
}

public Long getTransactionNumber()
{
    return transactionNumber;
}

public void setTransactionNumber(Long transactionNumber)
{

```

```
        this.transactionNumber = transactionNumber;
    }

    public Date getTransactionDate()
    {
        return transactionDate;
    }

    public void setTransactionDate(Date transactionDate)
    {
        this.transactionDate = transactionDate;
    }

    @Override
    public String toString()
    {
        return "Order [id=" + id + ", status=" + status + ", transactionNumber=" +
transactionNumber + ", transactionDate=" + transactionDate
        + ", customer=" + customer + ", orderItems=" + orderItems + "];"
    }
}
```

The order item is very similar and only includes the SKU of the product being ordered along with the order quantity:

```
package com.redhat.refarch.microservices.sales.model;

import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
@Entity
public class OrderItem
{

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private Long sku;
    private Integer quantity;
```

```
@ManyToOne(optional = false, fetch = FetchType.EAGER)
@JoinColumn(name = "ORDER_ID", referencedColumnName = "ID")
private Order order;

public Long getId()
{
    return id;
}

public void setId(Long id)
{
    this.id = id;
}

public Long getSku()
{
    return sku;
}

public void setSku(Long sku)
{
    this.sku = sku;
}

public Integer getQuantity()
{
    return quantity;
}

public void setQuantity(Integer quantity)
{
    this.quantity = quantity;
}

//Avoid getter so it is not included in automatic serialization
public Order retrieveOrder()
{
    return order;
}

public void setOrder(Order order)
{
    this.order = order;
}

@Override
public int hashCode()
```

```
{
    final int prime = 31;
    int result = 1;
    result = prime * result + ( ( id == null ) ? 0 : id.hashCode() );
    return result;
}

@Override
public boolean equals(Object obj)
{
    if( this == obj )
        return true;
    if( obj == null )
        return false;
    if( getClass() != obj.getClass() )
        return false;
    OrderItem other = (OrderItem)obj;
    if( id == null )
    {
        if( other.id != null )
            return false;
    }
    else if( !id.equals( other.id ) )
        return false;
    return true;
}

@Override
public String toString()
{
    return "OrderItem [id=" + id + ", sku=" + sku + ", quantity=" + quantity + "];"
}
}
```

Set up the database for the sales service similar to the product service. No sample data is required for this database to use the application:

```
CREATE DATABASE sales;

USE sales;
CREATE USER 'sales'@'localhost' IDENTIFIED BY 'password';
GRANT USAGE ON . TO 'sales'@'localhost' IDENTIFIED BY 'password';
GRANT ALL PRIVILEGES ON sales.* to sales@localhost;

CREATE TABLE CUSTOMER (ID BIGINT NOT NULL AUTO_INCREMENT, NAME VARCHAR(255) NOT NULL,
ADDRESS varchar(255), EMAIL varchar(255) NOT NULL, PASSWORD varchar(255), TELEPHONE
varchar(255), USERNAME varchar(255) NOT NULL UNIQUE, PRIMARY KEY (ID)) AUTO_INCREMENT =
100001;

CREATE TABLE ORDERS (ID BIGINT NOT NULL AUTO_INCREMENT, STATUS INTEGER, TRANSACTIONDATE
DATETIME, TRANSACTIONNUMBER BIGINT, CUSTOMER_ID BIGINT NOT NULL, PRIMARY KEY (ID))
AUTO_INCREMENT = 100001;

ALTER TABLE ORDERS ADD INDEX FK_ORDER_CUSTOMER (CUSTOMER_ID), add constraint
FK_ORDER_CUSTOMER FOREIGN KEY (CUSTOMER_ID) REFERENCES CUSTOMER (ID);

CREATE TABLE ORDERITEM (ID BIGINT NOT NULL AUTO_INCREMENT, SKU BIGINT NOT NULL, QUANTITY
INTEGER NOT NULL, ORDER_ID BIGINT NOT NULL, PRIMARY KEY (ID)) AUTO_INCREMENT = 1000001;

ALTER TABLE ORDERITEM ADD INDEX FK_ORDERITEM_ORDER (ORDER_ID), add constraint
FK_ORDERITEM_ORDER FOREIGN KEY (ORDER_ID) REFERENCES ORDERS (ID);
```

Database constraints provide further assurance that data integrity will be preserved, even if JPA is bypassed when entering or modifying data.

Copy the same *com.redhat.refarch.microservices.utils.Utils* class over to this project to use for full and partial updates to entities. The *com.redhat.refarch.microservices.sales.model.Error* class is also reused to map various business and runtime exceptions to RESTful error responses.

Create the sales service similar to the previously created RESTful Service for product:

```
package com.redhat.refarch.microservices.sales.service;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.net.HttpURLConnection;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.ejb.LocalBean;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.NoResultException;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;
import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.HttpMethod;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.WebApplicationException;
import com.redhat.refarch.microservices.sales.model.Customer;
import com.redhat.refarch.microservices.sales.model.Error;
import com.redhat.refarch.microservices.sales.model.Order;
import com.redhat.refarch.microservices.sales.model.Order.Status;
import com.redhat.refarch.microservices.sales.model.OrderItem;
import com.redhat.refarch.microservices.utils.Utills;

@Stateless
@LocalBean
@Path("/")
public class SalesService
{
    private Logger logger = Logger.getLogger( getClass().getName() );

    @PersistenceContext
    private EntityManager em;
```

Follow the same principles for RESTful Resource API design to create CRUD operations for customer. Only allow the search for customers based on their username:

```
@POST
@Path("/customers")
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Customer addCustomer(Customer customer)
{
    try
    {
        em.persist( customer );
        return customer;
    }
    catch( RuntimeException e )
    {
        throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR,
                        e ).asException();
    }
}

@GET
@Path("/customers")
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Customer getCustomer(@QueryParam("username") String username)
{
    try
    {
        TypedQuery<Customer> query = em.createNamedQuery(
            "Customer.findByUsername", Customer.class );
        Customer customer = query.setParameter(
            "username", username ).getSingleResult();
        logInfo( "Customer for " + username + ": " + customer );
        return customer;
    }
    catch( NoResultException e )
    {
        throw new Error( HttpURLConnection.HTTP_NOT_FOUND,
                        "Customer not found" ).asException();
    }
    catch( RuntimeException e )
    {
        throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR,
                        e ).asException();
    }
}

@GET
```

```
@Path("/customers/{id}")
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Customer getCustomer(@PathParam("id") Long id)
{
    try
    {
        logInfo( "Customer Id is " + id );
        Customer customer = em.find( Customer.class, id );
        logInfo( "Customer with ID " + id + " is " + customer );
        if( customer == null )
        {
            throw new Error( HttpURLConnection.HTTP_NOT_FOUND,
                            "Customer not found" ).asException();
        }
        return customer;
    }
    catch( WebApplicationException e )
    {
        throw e;
    }
    catch( RuntimeException e )
    {
        throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR,
                        e ).asException();
    }
}

@PUT
@Path("/customers/{id}")
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Customer updateCustomer(@PathParam("id") Long id,
                               Customer customer)
{
    Customer entity = getCustomer( id );
    try
    {
        //Ignore any attempt to update customer Id:
        customer.setId( id );
        Utils.copy( customer, entity, false );
        em.merge( entity );
        return entity;
    }
    catch( RuntimeException e )
    {
        throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR,
                        e ).asException();
    }
}
```

```
}
@PATCH
@Path("/customers/{id}")
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Customer partiallyUpdateCustomer(@PathParam("id") Long id,
                                       Customer customer)
{
    Customer entity = getCustomer( id );
    try
    {
        //Ignore any attempt to update customer Id:
        customer.setId( id );
        Utils.copy( customer, entity, true );
        em.merge( entity );
        return entity;
    }
    catch( RuntimeException e )
    {
        throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR,
                        e ).asException();
    }
}

@DELETE
@Path("/customers/{id}")
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public void deleteCustomer(@PathParam("id") Long id)
{
    Customer entity = getCustomer( id );
    try
    {
        em.remove( entity );
    }
    catch( RuntimeException e )
    {
        throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR,
                        e ).asException();
    }
}
```

5.4.10. Sub-resources, RESTful relationships

Orders are also modeled as resources, as described in the section on Resource API design, but an order only exists in the context of a customer and the REST API can reflect this fact by using the path to the parent customer as the relative context for the order.

The path for each REST operation therefore follows the same pattern, but is preceded by `/customers/{customerId}`:

```
@POST
@Path("/customers/{customerId}/orders")
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Order addOrder(@PathParam("customerId") Long customerId, Order order)
{
    Customer customer = getCustomer( customerId );
    order.setCustomer( customer );
    try
    {
        em.persist( order );
        return order;
    }
    catch( RuntimeException e )
    {
        throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR,
                        e ).asException();
    }
}

@GET
@Path("/customers/{customerId}/orders")
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public List<Order> getOrders(@PathParam("customerId") Long customerId,
@QueryParam("status") Status status)
{
    logInfo( "getOrders(" + customerId + ", " + status + ")" );
    Customer customer = getCustomer( customerId );
    try
    {
        TypedQuery<Order> query;
        if( status == null )
        {
            query = em.createNamedQuery( "Order.findByCustomer",
                                        Order.class );
        }
        else
        {
            query = em.createNamedQuery( "Order.findByOrderStatus",
                                        Order.class );
            query.setParameter( "status", status );
        }
        query.setParameter( "customer", customer );
        List<Order> orders = query.getResultList();
    }
}
```

```
        logInfo( "Orders retrieved as " + orders );
        return orders;
    }
    catch( RuntimeException e )
    {
        throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR,
                        e ).asException();
    }
}

@GET
@Path("/customers/{customerId}/orders/{orderId}")
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Order getOrder(@PathParam("customerId") Long customerId,
                    @PathParam("orderId") Long orderId)
{
    try
    {
        Order order = em.find( Order.class, orderId );
        logInfo( "Order retrieved as " + order );
        if( order != null && customerId.equals(
            order.retrieveCustomer().getId() ) )
        {
            return order;
        }
        else
        {
            throw new Error( HttpURLConnection.HTTP_NOT_FOUND,
                            "Order not found" ).asException();
        }
    }
    catch( WebApplicationException e )
    {
        throw e;
    }
    catch( RuntimeException e )
    {
        throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR,
                        e ).asException();
    }
}

@PUT
@Path("/customers/{customerId}/orders/{orderId}")
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Order updateOrder(@PathParam("customerId") Long customerId,
                        @PathParam("orderId") Long orderId, Order order)
```

```
{
    Order entity = getOrder( customerId, orderId );
    try
    {
        //Ignore any attempt to update order Id:
        order.setId( orderId );
        Utils.copy( order, entity, false );
        em.merge( entity );
        return entity;
    }
    catch( RuntimeException e )
    {
        throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR,
                        e ).asException();
    }
}

@PATCH
@Path("/customers/{customerId}/orders/{orderId}")
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Order partiallyUpdateOrder( @PathParam("customerId") Long customerId,
@PathParam("orderId") Long orderId, Order order)
{
    Order entity = getOrder( customerId, orderId );
    try
    {
        //Ignore any attempt to update order Id:
        order.setId( orderId );
        Utils.copy( order, entity, true );
        em.merge( entity );
        return entity;
    }
    catch( RuntimeException e )
    {
        throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR,
                        e ).asException();
    }
}

@DELETE
@Path("/customers/{customerId}/orders/{orderId}")
public void deleteOrder(@PathParam("customerId") Long customerId,
                        @PathParam("orderId") Long orderId)
{
    Order entity = getOrder( customerId, orderId );
    try
    {
```

```

        em.remove( entity );
    }
    catch( RuntimeException e )
    {
        throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR,
                        e ).asException();
    }
}

```

The order item is similarly treated as a resource underneath order. The path for its operations are therefore preceded by `/customers/{customerId}/orders/{orderId}/`:

```

@POST
@Path("/customers/{customerId}/orders/{orderId}/orderItems")
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public OrderItem addOrderItem(@PathParam("customerId") Long customerId,
                              @PathParam("orderId") Long orderId, OrderItem orderItem)
{
    Order order = getOrder( customerId, orderId );
    orderItem.setOrder( order );
    try
    {
        em.persist( orderItem );
        return orderItem;
    }
    catch( RuntimeException e )
    {
        throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR, e ).asException();
    }
}

@GET
@Path("/customers/{customerId}/orders/{orderId}/orderItems")
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public List<OrderItem> getOrderItems(@PathParam("customerId") Long customerId,
@PathParam("orderId") Long orderId)
{
    Order order = getOrder( customerId, orderId );
    if( order == null )
    {
        throw new Error( HttpURLConnection.HTTP_NOT_FOUND, "Order not found"
).asException();
    }
    try
    {
        return order.getOrderItems();
    }
}

```

```
}
catch( RuntimeException e )
{
    throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR, e ).asException();
}

}

@GET
@Path("/customers/{customerId}/orders/{orderId}/orderItems/{orderId}")
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public OrderItem getOrderItem(@PathParam("customerId") Long customerId,
    @PathParam("orderId") Long orderId,
    @PathParam("orderId") Long orderId)
{
    try
    {
        OrderItem orderItem = em.find( OrderItem.class,
            orderId );

        if( orderItem != null && orderId.equals( orderItem.retrieveOrder().getId() )
            && customerId.equals( orderItem.retrieveOrder().retrieveCustomer().getId() )
)
        {
            return orderItem;
        }
        else
        {
            throw new Error( HttpURLConnection.HTTP_NOT_FOUND,
                "Order Item not found" ).asException();
        }
    }
    catch( WebApplicationException e )
    {
        throw e;
    }
    catch( RuntimeException e )
    {
        throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR, e ).asException();
    }
}

@PUT
@Path("/customers/{customerId}/orders/{orderId}/orderItems/{orderId}")
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public OrderItem updateOrderItem(
    @PathParam("customerId") Long customerId,
    @PathParam("orderId") Long orderId,
    @PathParam("orderId") Long orderId, OrderItem orderItem)
```

```
{
    OrderItem entity = getOrderItem( customerId, orderId, orderItemId );
    try
    {
        //Ignore any attempt to update order item Id:
        orderItem.setId( orderItemId );
        Utils.copy( orderItem, entity, false );
        em.merge( entity );
        return entity;
    }
    catch( RuntimeException e )
    {
        throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR, e ).asException();
    }
}

@PATCH
@Path("/customers/{customerId}/orders/{orderId}/orderItems/{orderItemId}")
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public OrderItem partiallyUpdateOrderItem(
    @PathParam("customerId") Long customerId,
    @PathParam("orderId") Long orderId,
    @PathParam("orderItemId") Long orderItemId, OrderItem orderItem)
{
    OrderItem entity = getOrderItem( customerId, orderId, orderItemId );
    try
    {
        //Ignore any attempt to update order item Id:
        orderItem.setId( orderItemId );
        Utils.copy( orderItem, entity, true );
        em.merge( entity );
        return entity;
    }
    catch( RuntimeException e )
    {
        throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR, e ).asException();
    }
}

@DELETE
@Path("/customers/{customerId}/orders/{orderId}/orderItems/{orderItemId}")
public void deleteOrderItem(@PathParam("customerId") Long customerId,
    @PathParam("orderId") Long orderId,
    @PathParam("orderItemId") Long orderItemId)
{
    Order order = getOrder( customerId, orderId );
    OrderItem entity = getOrderItem( customerId, orderId, orderItemId );
}
```

```
try
{
    em.remove( entity );
    order.getOrderItems().remove( entity );
    em.merge( order );
}
catch( RuntimeException e )
{
    throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR, e ).asException();
}
```

The only other operation in the sales service is an authenticate method that verifies the customer's credentials:

```
@POST
@Path("/authenticate")
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Customer authenticate(Customer customer)
{
    logInfo( "Asked to authenticate " + customer );
    Customer response = getCustomer( customer.getUsername() );
    try
    {
        if( response.getPassword().equals( customer.getPassword() ) == false )
        {
            throw new WebApplicationException( HttpURLConnection.HTTP_UNAUTHORIZED );
        }
        return response;
    }
    catch( WebApplicationException e )
    {
        throw e;
    }
    catch( RuntimeException e )
    {
        throw new Error( HttpURLConnection.HTTP_INTERNAL_ERROR, e ).asException();
    }
}
```

The security aspect of this reference application is only for demonstration purposes. This authentication example is not intended to provide a reference of security best practices and the security aspect of these uses cases is beyond the scope of this reference architecture.

This service class also uses the JDK logging framework and declares the PATCH annotation for partial updates:

```
private void logInfo(String message)
{
    logger.log( Level.INFO, message );
}

@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@HttpMethod("PATCH")
public @interface PATCH
{
}
}
```

5.4.11. Billing Service

This reference architecture assumes the existence of a third microservice to process credit card transactions. This functionality is provided by an external system with a legacy interface, wrapped by a microservice that exposes a REST API.

Once again, either create a new JBDS project called Billing and follow the same steps that were used to create the product project, or optionally, you can also duplicate the existing product project in JBoss Developer Studio, renaming it to billing and making the necessary changes. This project does not require database and JPA dependencies.

The service models its request as a transaction object:

```
@XmlElement
public class Transaction
{

    private Long creditCardNumber;
    private Integer expMonth;
    private Integer expYear;
    private Integer verificationCode;
    private String billingAddress;
    private String customerName;
    private Long orderNumber;
    private Double amount;

    ...
}
```

A result type is defined to provide a response to a transaction processing request:

```
@XmlRootElement
public class Result
{

    public enum Status
    {
        SUCCESS, FAILURE
    }

    private Status status;
    private String name;
    private Long orderNumber;
    private Date transactionDate;
    private Integer transactionNumber;

    ...
}
```

5.4.12. Asynchronous Processing

The main operation of the billing service receives a transaction request, processes it and depending on the outcome, returns the appropriate result. The legacy credit card processing system can be slow and consumes resources on a different thread or even different system, so this service is a prime candidate for asynchronous processing. The *process* operation injects an *AsyncResponse* and uses the [executor service](#) to call a synchronous operation on a different thread:

```
@Path("/")
public class BillingService {

    private ManagedExecutorService executorService;

    private Logger logger = Logger.getLogger( getClass().getName() );

    private static final Random random = new Random();

    @POST
    @Path("/process")
    @Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
    @Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
    public void process(final Transaction transaction, final @Suspended AsyncResponse
asyncResponse) {

        Runnable runnable = () -> {
            try {
                final long sleep = 5000;
                logInfo( "Will simulate credit card processing for " + sleep + "
milliseconds" );
                Thread.sleep( sleep );
                Result result = processSync( transaction );
                asyncResponse.resume( result );
            } catch (Exception e) {
                asyncResponse.resume( e );
            }
        };
        getExecutorService().execute( runnable );
    }
}
```

This service simulates processing delays by putting the thread to sleep for 5 seconds. After waking up, the asynchronous response object is used to return the result.

For this mock-up service, the success of the credit card transaction only depends on the credit card having a future expiration date. The transaction number is randomly generated as a 7-digit number and the transaction date matches the time of the request.

```
private Result processSync(Transaction transaction) {
    Result result = new Result();
    result.setName( transaction.getCustomerName() );
    result.setOrderNumber( transaction.getOrderNumber() );
    logInfo( "Asked to process credit card transaction: " + transaction );
    Calendar now = Calendar.getInstance();
    Calendar calendar = Calendar.getInstance();
    calendar.clear();
    calendar.set( transaction.getExpYear(), transaction.getExpMonth(), 1 );
    if( calendar.after( now ) ) {
        result.setTransactionNumber( (long)(random.nextInt( 9000000 ) + 1000000) );
        result.setTransactionDate( now.getTime() );
        result.setStatus( Status.SUCCESS );
    } else {
        result.setStatus( Status.FAILURE );
    }
    return result;
}
```

Another operation is provided to reverse the transaction charge:

```
@POST
@Path("/refund/{transactionNumber}")
@Consumes({"/"})
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public void refund(@PathParam("transactionNumber") int transactionNumber)
{
    logInfo( "Asked to refund credit card transaction: " + transactionNumber );
}
```

5.4.13. Managed Executor Service

The billing service uses a Java EE 7 managed executor service to perform the actual credit card processing on a separate managed thread. To use this library in your project, first declare the necessary maven dependency:

```
<dependencies>
...
<dependency>
  <groupId>org.jboss.spec.javax.enterprise.concurrent</groupId>
  <artifactId>jboss-concurrency-api_1.0_spec</artifactId>
  <scope>provided</scope>
</dependency>
...
```

This allows `ManagedExecutorService` to be imported:

```
private ManagedExecutorService executorService;
```

In certain circumstances, resource injection may be used to instantiate the executor service. However in this case and in a *Servlet* that has not been added to the *ServletContext*, simply use a JNDI lookup:

```
private Executor getExecutorService() {
    if( executorService == null ) {
        try {
            executorService = InitialContext.doLookup(
"java:comp/DefaultManagedExecutorService" );
        } catch (NamingException e) {
            throw new WebApplicationException( e );
        }
    }
    return executorService;
}
```

While this example uses a default and generic executor service, a custom managed executor service may be defined and configured for specific requirements. Refer to the [official Red Hat documentation on this topic](#) for further details.

5.5. Aggregation/Presentation Layer

This reference application uses a JSP layer to both generate client-side HTML, and act as the aggregator layer for the architecture.

The product, sales and billing services in this reference architecture must exist within a trust perimeter. This JSP layer acts as the aggregation layer and is the only client permitted to directly access these services and can coordinate the response from each service as required.

The aggregation layer uses the **JAX-RS 2.0** client library to communicate with the three microservices. Through Maven dependencies, this layer is able to directly access the data model used by each service. To achieve this, the maven build file of each of those three services is configured to generate a JAR file of the classes, in addition to the deployable WAR file:

```
<plugin>
  <artifactId>maven-war-plugin</artifactId>
  <version>${version.war.plugin}</version>
  <configuration>
    <!-- Java EE 7 doesn't require web.xml, Maven needs to catch up! -->
    <failOnMissingWebXml>false</failOnMissingWebXml>
    <attachClasses>true</attachClasses>
  </configuration>
</plugin>
```

This results in a JAR file that the aggregation layer can declare as a dependency:

```
<dependencies>
  <dependency>
    <groupId>com.redhat.refarch.microservices</groupId>
    <artifactId>product</artifactId>
    <version>1.0</version>
    <classifier>classes</classifier>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>com.redhat.refarch.microservices</groupId>
    <artifactId>sales</artifactId>
    <version>1.0</version>
    <classifier>classes</classifier>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>com.redhat.refarch.microservices</groupId>
    <artifactId>billing</artifactId>
    <version>1.0</version>
    <classifier>classes</classifier>
    <scope>compile</scope>
  </dependency>
  ...
```

The **JAX-RS 2.0** client library allows an application to communicate with services through the direct use of these objects.

One challenge arising from this dependency declaration is that by including the *product* and *sales* JAR modules, the aggregation layer is also unintentionally deploying their respective persistence units. To prevent JBoss EAP from attempting to deploy persistence units and connect to a datasource on the presentation layer, a simple workaround is to exclude JPA capabilities by including a deployment descriptor at `src/main/webapp/WEB-INF/jboss-deployment-structure.xml`. This descriptor may explicitly exclude a subsystem from the containing module:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-deployment-structure>
  <deployment>
    <exclude-subsystems>
      <subsystem name="jpa" />
    </exclude-subsystems>
  </deployment>
</jboss-deployment-structure>
```

To easily access the microservices from JSPs, create a utility class with convenience methods to call various operations on the services and orchestrate them as necessary:

```
package com.redhat.refarch.microservices.presentation;

import ...

public class RestClient
{
    private enum Service
    {
        Product, Sales, Billing
    };
}
```

Create a convenience method to help specify the address for each service call:

```
private static WebTarget getWebTarget(Service service, Object... path) {
    Client client = ClientBuilder.newClient();
    WebTarget target;
    switch (service) {
        case Product:
            target = client.target("http://product-service").path("/product");
            break;

        case Sales:
            target = client.target("http://sales-service").path("/sales");
            break;

        case Billing:
            target = client.target("http://billing-service").path("/billing");
            break;

        default:
            throw new IllegalStateException("Unknown service");
    }
    for (Object part : path) {
        target = target.path(String.valueOf(part));
    }
    return target;
}
```

This method assumes that the *product*, *sales* and *billing* service are respectively accessible through the host names of *product-service*, *sales-service* and *billing-service*. It further assumes a root context for each of these services and the port as 8080. Using **Apache HTTP Server** or another load balancer, the root context may be unnecessary and the standard port of 80 is more likely to be used.

The remaining parts of the service address are passed to this method as arguments and used to construct the remainder of the service URL path.

With the help of this convenience method, to retrieve a list of featured products:

```
private static List<Product> getFeaturedProducts() throws HttpException {
    WebTarget webTarget = getWebTarget(Service.Product,
    "products").queryParam("featured", true);
    logInfo("Executing " + webTarget.getUri());
    Response response = webTarget.request(MediaType.APPLICATION_JSON).get();
    if (isError(response)) {
        throw new HttpException(response);
    } else {
        return response.readEntity(new GenericType<List<Product>>() {
        });
    }
}
```

Notice that query parameters can still be added to the *WebTarget* after it is returned from the convenience method.

After executing the call, it is important to check the HTTP response status code for errors. If there is an error, throw an exception that includes the error code and description. If the call is successful, parse each JSON object to the corresponding data model and return class type.

To determine if there is an error, use a convenience method that checks for HTTP status codes 400 and greater:

```
private static boolean isError(Response response) {
    if (response.getStatus() >= HttpStatus.SC_BAD_REQUEST) {
        return true;
    } else {
        return false;
    }
}
```

While most errors will manifest themselves with a standard HTTP error code and a descriptive message, there will also be system errors in either the services or the client that cannot be anticipated. These errors are reported as standard Java exceptions and for consistency, you can also use a Java exception to report known HTTP errors:

```
package com.redhat.refarch.microservices.presentation;

import javax.ws.rs.ProcessingException;
import javax.ws.rs.core.Response;

public class HttpErrorException extends Exception {

    private static final long serialVersionUID = 1L;
    private int code;
    private String content;

    public HttpErrorException(Response response) {
        code = response.getStatus();
        try {
            content = response.readEntity(String.class);
        } catch (ProcessingException | IllegalStateException e) {
            content = "Unknown";
        }
    }

    @Override
    public String getMessage() {
        return "HTTP Error " + code + ": " + content;
    }
}
```

Searching for products based on one or more keywords is very similar and takes advantage of the same convenience method:

```
private static List<Product> searchProducts(String query) throws HttpException {
    WebTarget webTarget = getWebTarget(Service.Product, "products");
    for (String keyword : query.split("\\s+")) {
        webTarget = webTarget.queryParam("keyword", keyword);
    }
    logInfo("Executing " + webTarget.getUri());
    Response response = webTarget.request(MediaType.APPLICATION_JSON).get();
    if (isError(response)) {
        throw new HttpException(response);
    } else {
        return response.readEntity(new GenericType<List<Product>>() {
        });
    }
}
```

Any whitespaces found in the query are treated as separators between multiple keywords. The keywords are passed to the service as a multi-valued query parameter.

Create a convenience method to retrieve products and set the result as a request attribute:

```
public static void setProductsAttribute(HttpServletRequest request) {
    try {
        List<Product> products;
        String query = request.getParameter("query");
        if (query == null || query.isEmpty()) {
            products = getFeaturedProducts();
        } else {
            products = searchProducts(query);
        }
        request.setAttribute("products", products);
    } catch (Exception e) {
        e.printStackTrace();
        request.setAttribute("errorMessage", "Failed to retrieve products: " +
e.getMessage());
    }
}
```

Notice that in case of an error, the *errorMessage* request attribute is set.

Remember that the presentation layer of this reference architecture is only provided for demo purposes and not intended to convey best practices in terms of the design and development of web applications and presentation layers. As such, create a simple JSP to display the products. This will be the main driver of the presentation layer. Place this file in the top directory of the web application and call it *index.jsp*:

```
<%@page
    import="com.redhat.refarch.microservices.presentation.RestClient"%>
<%@page import="java.util.Map"%>
<%@page import="java.util.List"%>
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Products</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
</head>
<body>
    <c:choose>
        <c:when test="${param.register}">
            <%@include file="register.jsp"%>
        </c:when>
        <c:when test="${param.cart}">
            <%@include file="cart.jsp"%>
        </c:when>
        <c:when test="${param.checkout}">
            <%@include file="checkout.jsp"%>
        </c:when>
        <c:when test="${param.history}">
            <%@include file="history.jsp"%>
        </c:when>
        <c:otherwise>
            <c:choose>
                <c:when test="${param.purchase}">
                    <%
                        RestClient.purchase( request );
                    %>
                </c:when>
                <c:when test="${param.registration}">
                    <%
                        RestClient.register( request );
                    %>
                </c:when>
            </c:choose>
        </c:otherwise>
    </c:choose>
</body>
</html>
```

```
        %>
    </c:when>
    <c:when test="${param.login}">
        <%
            RestClient.login( request );
        %>
    </c:when>
    <c:when test="${param.logout}">
        <%
            RestClient.logout( request );
        %>
    </c:when>
    <c:when test="${param.completeOrder}">
        <%
            RestClient.completeOrder( request );
        %>
    </c:when>
</c:choose>
<%
    RestClient.setProductsAttribute( request );
%>

<%@include file="header.jsp"%>

<%@include file="products.jsp"%>
</c:otherwise>
</c:choose>
</body>
</html>
```

This *index.jsp* file gets resolved on every request. It uses the core tag library provided in JSTL for conditional behavior and in case of the presence of certain request parameters, namely *register*, *cart*, *checkout*, and *history*, the product list is not displayed and instead, the corresponding JSP is used. In these cases, the index file simply delegates to another JSP.

When none of these four parameters are included in the request, this JSP still checks for 5 other request parameters, called *purchase*, *registration*, *login*, *logout*, and *completeOrder*. While each of these parameters results in a method invocation on the same *RestClient* class, they do not impact further processing of *index.jsp* and also result in products being listed on the page. This is done through a call to *RestClient.setProductsAttribute(request)* so that the products are retrieved and stored as a request attributed, followed by the inclusion of *products.jsp*, which renders the list of products as HTML.

The *products.jsp* file is a simple JSP with two main functions. It provides a search box to allow the user to search for products based on one or multiple keywords:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<div style="margin-top: 5em;"></div>
<form target="_self" method="post" style="margin: 0px auto;">
  <table style="margin: 0px auto; width: 30em; border: 0px;">
    <tr>
      <td><input type="text" name="query" size="50">
        <button type="submit">Search</button></td>
    </tr>
  </table>
</form>
```

This JSP also uses the core tag library to loop through products stored as a request attribute, if any are stored, and display them as HTML:

```
<c:forEach var="product" items="${products}">
  <br />
  <br />
  <table style="margin: 0px auto; width: 80%; border: 1px solid black;">
    <caption style="margin: 0px auto; font-size: 3em">
      ${product.name}</caption>
    <tr style="border: 1px solid black;">
      <td style="border: 1px solid black; padding: 5px">
        </td>
      <td style="border: 1px solid black; padding: 5px">
        ${product.description}</td>
      <td style="border: 1px solid black; padding: 5px">
        Product Dimensions:
        ${product.length} x ${product.width} x ${product.height}
        <br />
        Product Weight: ${product.weight}
      </td>
      <td style="border: 1px solid black; padding: 5px">
        <p style="font-size: 1.5em">${product.price}</p>
        <p>Availability: ${product.availability}</p>

        <c:if test="${sessionScope.customer != null}">
          <form target="_self" method="post">
            <input type="hidden" name="sku" value="${product.sku}">
            <button name="purchase" value="true" type="submit">
              Purchase</button>
          </form>
        </c:if>
      </td>
    </tr>
  </table>
</c:forEach>
```

This code also checks to see if the customer is logged in, indicated by the presence of a customer object in the HTTP session. If the customer is in fact logged in, a *purchase* button is also provided for each listed product.

Pressing the purchase button submits the form and reinvokes the same *index.jsp* file, but this time, the purchase request parameter will be present (as the name of the button) and have a value of true. This will trigger the invocation of *RestClient.purchase(request)*.

Note that static resources are assumed to be served from a location accessible through */images*.

The header JSP provides a login bar that includes a register button for new customers. On the left and at the beginning of this bar, a generic placeholder is included for any message informing the user of success or failure. The request parameter potentially includes such a success or failure message and is rendered in green or red color:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<form id="headerForm" target="_self" method="post">
  <table style="width: 100%;">
    <tr>
      <c:if test="${not empty successMessage}">
        <td>
          <div style="color: green">${successMessage}</div>
        </td>
      </c:if>
      <c:if test="${not empty errorMessage}">
        <td>
          <div style="color: red">${errorMessage}</div>
        </td>
      </c:if>
    </tr>
  </table>
</form>
```

If the user is logged in (customer object is present in the HTTP session), they are presented with a welcome message that includes their name, as retrieved from the database and returned in the customer object. There is also a hyperlink that takes the customer to their order history page, by calling a JavaScript method at the end of this JSP file, which sets the value of the hidden history input to true and submits the form.

The logout button also submits the form, while including a logout request parameter (button name) with a value of true, that will result in a call to log the user out:

```
<c:if test="${not empty sessionScope.customer}">
  <td>
    <table style="float: right; border: 0; text-align: right;">
      <tr>
        <td style="margin-right: 20px;">Welcome back, ${customer.name}</td>
        <td style="padding-right: 20px; padding-left: 20px;">
          <a href="javascript:" onclick="history();">Order History</a>
        </td>
      </tr>
      <tr>
        <td>
          <button name="logout" value="true"
            style="margin-right: 20px; margin-left: 20px;">Log Out</button>
        </td>
      </tr>
    </table>
  </td>
</c:if>
```

If the customer has any items in their shopping cart, the shopping cart icon is displayed with an opacity of 0.6, and the number of items in the cart is superimposed on top of the cart icon. Clicking on either the cart or the number takes the customer to their shopping cart, by using the *clickCart()* JavaScript method, which simply clicks the hidden cart button.

```
<c:if test="${itemCount > 0}">
  <td><button name="cart" id="cart" value="true"
    style="visibility: hidden;"></button></td>
  <td style="margin-right: 10px; display: block; position: relative;">
    
    <p style="opacity: 1; position: absolute; top: 0; left: 15px;"
      onclick="clickCart();">
      <c:out value="${itemCount}" />
    </p>
  </td>
</c:if>
```

If the shopping cart is empty, the icon is displayed in full opacity and clicking on the icon is disabled:

```
<c:if test="${itemCount == 0}">
  <td
    style="margin-right: 10px; display: block; position: relative;">
    
  </td>
</c:if>
</tr>
</table>
</td>
</c:if>
```

If the customer is not logged in, user and password input fields are provided with a login button for existing customers to log in, as well as a register button for new customers. Again, any of these actions submits the form with appropriate request parameters that lead the *index.jsp* file to react properly.

The simple JavaScript functions are provided at the end of the header file:

```
<script type="text/javascript">
  function history() {
    document.getElementById('history').value = true;
    document.getElementById("headerForm").submit();
  }
  function clickCart() {
    document.getElementById('cart').click();
  }
</script>
```

When a user clicks the register button, the corresponding JSP renders a form that can be filled out and submitted to register a new customer:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<form target="_self" method="post">
  <table style="margin: 0px auto; border: 1px solid black;">
    <caption style="margin: 0px auto; font-size: 2em">Customer Registration</caption>
    <tr style="border: 1px solid black;">
      <td style="border: 1px solid black; padding: 5px; min-width: 8em;">Name:</td>
      <td style="border: 1px solid black; padding: 5px;"><input
        name="name" type="text" size="30" /></td>
    </tr>
    <tr style="border: 1px solid black;">
      <td style="border: 1px solid black; padding: 5px; min-width: 8em;">Address:</td>
      <td style="border: 1px solid black; padding: 5px;"><input name="address"
type="text" size="30" /></td>
    </tr>
    <tr style="border: 1px solid black;">
      <td style="border: 1px solid black; padding: 5px; min-width:
8em;">Telephone:</td>
      <td style="border: 1px solid black; padding: 5px;"><input
        name="telephone" type="text" size="30" /></td>
    </tr>
    <tr style="border: 1px solid black;">
      <td style="border: 1px solid black; padding: 5px; min-width: 8em;">Email:</td>
      <td style="border: 1px solid black; padding: 5px;"><input
        name="email" type="text" size="30" /></td>
    </tr>
    <tr style="border: 1px solid black;">
      <td style="border: 1px solid black; padding: 5px; min-width: 8em;">Username:</td>
      <td style="border: 1px solid black; padding: 5px;"><input
        name="username" type="text" size="30" /></td>
    </tr>
    <tr style="border: 1px solid black;">
      <td style="border: 1px solid black; padding: 5px; min-width: 8em;">Password:</td>
      <td style="border: 1px solid black; padding: 5px;"><input
        name="password" type="password" size="30" /></td>
    </tr>
  </table>
  <div style="margin: 10px auto; width: 100%; text-align: center;">
    <button name="registration" value="true">Register</button>
    <button name="registration" value="false">Cancel</button>
  </div>
</form>
```

Note that the *Register* and *Cancel* buttons both set a request parameter called *registration*, but the value will be *true* or *false* depending on which button is pressed. When the value of this parameter is true, the index JSP calls the register method:

```
public static void register(HttpServletRequest request) throws ServletException {
    WebTarget webTarget = getWebTarget(Service.Sales, "customers");
    Customer customer = Utils.getRegistrationInfo(request);
    Entity<Customer> entity = Entity.entity(customer, MediaType.APPLICATION_JSON);
    logInfo("Executing " + webTarget.getUri() + " with " + customer);
    Response response = webTarget.request(MediaType.APPLICATION_JSON).post(entity);
    if (isError(response)) {
        request.setAttribute("errorMessage", "Failed to register customer");
    } else {
        customer = response.readEntity(Customer.class);
        logInfo("Got " + customer);
        request.getSession().setAttribute("customer", customer);
        request.getSession().setAttribute("itemCount", 0);
        getPendingOrder(request, customer.getId());
    }
}
```

The *register* method uses a convenience method in *Utils* to read the specified request parameters and create a *Customer* object based on their names and values:

```
public static Customer getRegistrationInfo(HttpServletRequest request) {
    Customer customer = new Customer();
    customer.setName(request.getParameter("name"));
    customer.setAddress(request.getParameter("address"));
    customer.setTelephone(request.getParameter("telephone"));
    customer.setEmail(request.getParameter("email"));
    customer.setUsername(request.getParameter("username"));
    customer.setPassword(request.getParameter("password"));
    return customer;
}
```

This method then proceed to use the object to post JSON to the /customers operation of the sales service. In case of an error response, an appropriate error message is placed in a request attribute and subsequently displayed to the user by the header file.

When the call to register a new customer is successful, the customer object is stored in the user's HTTP session and repeatedly check to determine if the user is logged in. Since the customer just registered, the number of items in the customer shopping cart is also initialized to a value of zero.

The login operation follows a similar path:

```
public static void login(HttpServletRequest request) throws HttpException {
    WebTarget webTarget = getWebTarget(Service.Sales, "authenticate");
    Customer customer = Utils.getLoginInfo(request);
    Entity<Customer> entity = Entity.entity(customer, MediaType.APPLICATION_JSON);
    logInfo("Executing " + webTarget.getUri() + " with " + customer);
    Response response = webTarget.request(MediaType.APPLICATION_JSON).post(entity);
    if (isError(response)) {
        int responseCode = response.getStatus();
        if (responseCode == HttpStatus.SC_UNAUTHORIZED) {
            request.setAttribute("errorMessage", "Incorrect password");
        } else if (responseCode == HttpStatus.SC_NOT_FOUND) {
            request.setAttribute("errorMessage", "Customer not found");
            request.setAttribute("username", customer.getUsername());
        } else {
            request.setAttribute("errorMessage", "Failed to login");
        }
    } else {
        customer = response.readEntity(Customer.class);
        logInfo("Got login response " + customer);
        request.getSession().setAttribute("customer", customer);
        request.getSession().setAttribute("itemCount", 0);
        getPendingOrder(request, customer.getId());
    }
}
```

When login succeeds, another method is called to find any potential items in the user's shopping cart. This application persists the content of the shopping cart in the database as an order with a status of *Initial*. The web application creates its own *Order* and *OrderItem* **JavaBean** types as needed:

```
package com.redhat.refarch.microservices.presentation;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;

public class Order
{
    private long id;
    private String status;
    private Long transactionNumber;
    private Date transactionDate;
    private List<OrderItem> orderItems = new ArrayList<OrderItem>();

    public long getId()
    {
        return id;
    }

    public void setId(long id)
    {
        this.id = id;
    }

    public String getStatus()
    {
        return status;
    }

    public void setStatus(String status)
    {
        this.status = status;
    }

    ...
    ...
}
```

The OrderItem type represents both a product and a sales order item, as understood by this later:

```
package com.redhat.refarch.microservices.presentation;

public class OrderItem
{
    private long id;
    private long sku;
    private int quantity;
    private String name;
    private String description;
    private Integer length;
    private Integer width;
    private Integer height;
    private Integer weight;
    private Boolean featured;
    private Integer availability;
    private Double price;
    private String image;

    public long getId()
    {
        return id;
    }

    public void setId(long id)
    {
        this.id = id;
    }

    public long getSku()
    {
        return sku;
    }

    public void setSku(long sku)
    {
        this.sku = sku;
    }
    ...
    ...
}
```

The method to retrieve pending orders queries the sales service:

```
private static void getPendingOrder(HttpServletRequest request, long customerId) {
    WebTarget webTarget = getWebTarget(Service.Sales, "customers", customerId,
    "orders").queryParam("status",
        Status.Initial.name());
    logInfo("Executing " + webTarget.getUri());
    Response response = webTarget.request(MediaType.APPLICATION_JSON).get();
    if (isError(response)) {
        logInfo("Failed to get pending order: " + response.readEntity(String.class));
    } else {
        List<Order> orders = response.readEntity(new GenericType<List<Order>>() {
        });
        logInfo("Got orders " + orders);
        if (orders.isEmpty()) {
            request.getSession().removeAttribute("orderId");
            request.getSession().removeAttribute("orderItems");
            request.getSession().setAttribute("itemCount", 0);
            request.removeAttribute("cart");
        } else {
            // Not expecting more than one pending order at a time:
            Order order = orders.get(0);
            request.getSession().setAttribute("orderId", order.getId());
            request.getSession().setAttribute("orderItems", order.getOrderItems());
            int cartSize = 0;
            try {
                updateInventory(request, order.getOrderItems());
                for (OrderItem orderItem : order.getOrderItems()) {
                    cartSize += orderItem.getQuantity();
                }
            } catch (HttpErrorException e) {
                logInfo("Failed to update inventory: " + e.getMessage());
                e.printStackTrace();
            }
            request.getSession().setAttribute("itemCount", cartSize);
            if (cartSize == 0) {
                request.removeAttribute("cart");
            }
        }
    }
}
```

The sales service only includes the product SKU. The method above uses the *updateInventory* convenience method to retrieve product details using the SKU and maintain a map from the *OrderItem* sku to product details:

```
private static void updateInventory(HttpServletRequest request, List<OrderItem>
orderItems)
    throws HttpException {
    @SuppressWarnings("unchecked")
    Map<Long, Product> inventory = (Map<Long, Product>)
request.getSession().getAttribute("inventory");
    if (inventory == null) {
        inventory = new HashMap<>();
    }
    for (OrderItem orderItem : orderItems) {
        Product product = lookupProduct(orderItem.getSku());
        inventory.put(product.getSku(), product);
    }
    request.getSession().setAttribute("inventory", inventory);
}
```

Logging out a customer in response to a click of the logout button is simply a matter of clearing the session content:

```
public static void logout(HttpServletRequest request)
{
    HttpSession session = request.getSession();
    Enumeration<String> attrNames = session.getAttributeNames();
    while( attrNames.hasMoreElements() )
    {
        session.removeAttribute( attrNames.nextElement() );
    }
}
```

The purchase operation, in response to the purchase button for a product, is one of the more complicated tasks.

This method first gets the product inventory information and checks the availability of the product. If not available, an error message is returned to the customer. When there is sufficient availability, the customer information is retrieved and the order ID for the pending order, representing the customer's shopping cart content, is requested.

If no initial order exists, one is created for the customer, effectively creating a new shopping cart, and adding the selected product with a quantity of one, as the first order:

```
public static void purchase(HttpServletRequest request) throws ServletException {
    long sku = Long.valueOf(request.getParameter("sku"));
    int availability = lookupProduct(sku).getAvailability();
    if (availability == 0) {
        request.setAttribute("errorMessage", "The selected item is not available for purchase!");
        return;
    }
    Customer customer = (Customer) request.getSession().getAttribute("customer");
    long customerId = customer.getId();
    Long orderId = (Long) request.getSession().getAttribute("orderId");
    if (orderId == null) {
        orderId = addInitialOrder(customerId);
        addOrderItem(customerId, orderId, sku, 1);
    }
}
```

If a shopping cart already exists for this customer, the order items in the shopping cart are retrieved and searched using Java 8 list stream filters for the product being purchased. If the product was not previously ordered, it is again added as a new order item with a quantity of one. If the product is found in the customer shopping cart and has previously been ordered, `updateOrderItem` is used to request that the quantity of that order be increased by one count.

Finally, the updated shopping cart data is requested from the server to make sure it reflects the correct information:

```
else {
    @SuppressWarnings("unchecked")
    List<OrderItem> orderItems = (List<OrderItem>)
request.getSession().getAttribute("orderItems");
    Optional<OrderItem> optionalItem = orderItems.stream().filter(x ->
x.getSku().equals(sku)).findFirst();
    if (optionalItem.isPresent()) {
        OrderItem orderItem = optionalItem.get();
        long orderItemId = orderItem.getId();
        int quantity = orderItem.getQuantity() + 1;
        updateOrderItem(request, customerId, orderId, orderItemId, sku, quantity);
    } else {
        addOrderItem(customerId, orderId, sku, 1);
    }
}
getPendingOrder(request, customerId);
}
```

To get the available inventory count for the product, the product microservice is used to look up the product details:

```
private static Product lookupProduct(Long sku) throws HttpErrorException {
    WebTarget webTarget = getWebTarget(Service.Product, "products", sku);
    logInfo("Executing " + webTarget.getUri());
    Response response = webTarget.request(MediaType.APPLICATION_JSON).get();
    if (isError(response)) {
        throw new HttpErrorException(response);
    } else {
        Product product = response.readEntity(Product.class);
        logInfo("Product looked up as " + product);
        return product;
    }
}
```

To create a new shopping cart by adding an initial order, simply post an order object that has a status of *Initial*:

```
private static long addInitialOrder(long customerId) throws HttpErrorException {
    WebTarget webTarget = getWebTarget(Service.Sales, "customers", customerId, "orders");
    Order order = new Order();
    order.setStatus(Status.Initial);
    Entity<Order> entity = Entity.entity(order, MediaType.APPLICATION_JSON);
    logInfo("Executing " + webTarget.getUri() + " with " + order);
    Response response = webTarget.request(MediaType.APPLICATION_JSON).post(entity);
    if (isError(response)) {
        HttpErrorException exception = new HttpErrorException(response);
        logInfo("Failed to add initial order: " + exception.getMessage());
        throw exception;
    } else {
        Order initialOrder = response.readEntity(Order.class);
        logInfo("Stored initial order as " + initialOrder);
        return initialOrder.getId();
    }
}
```

Adding an order item to a now-existing order is also straight-forward and uses the resource REST API where an order item exists withing an order, which exists within a customer:

```
private static long addOrderItem(long customerId, long orderId, long sku, int quantity)
throws HttpErrorException {
    WebTarget webTarget = getWebTarget(Service.Sales, "customers", customerId, "orders",
orderId, "orderItems");
    OrderItem orderItem = new OrderItem();
    orderItem.setSku(sku);
    orderItem.setQuantity(quantity);
    Entity<OrderItem> entity = Entity.entity(orderItem, MediaType.APPLICATION_JSON);
    logInfo("Executing " + webTarget.getUri() + " with " + orderItem);
    Response response = webTarget.request(MediaType.APPLICATION_JSON).post(entity);
    if (isError(response)) {
        HttpErrorException exception = new HttpErrorException(response);
        logInfo("Failed to add order item: " + exception.getMessage());
        throw exception;
    } else {
        OrderItem addedOrderItem = response.readEntity(OrderItem.class);
        logInfo("Added order item: " + addedOrderItem);
        return addedOrderItem.getId();
    }
}
```

To update the order quantity for an order item, first verify that there is available inventory and if there isn't, set an appropriate error message. Then use a partial update to only update the quantity of the order item:

```
private static void updateOrderItem(HttpServletRequest request, long customerId, long
orderId, long orderItemId,
    Long sku, int quantity) throws HttpErrorException {
    if (sku == null) {
        sku = lookupOrderItem(customerId, orderId, orderItemId).getSku();
    }
    int availability = lookupProduct(sku).getAvailability();
    if (quantity > availability) {
        quantity = availability;
        request.setAttribute("errorMessage", "Requested quantity exceeds product
availability");
    }
    WebTarget webTarget = getWebTarget(Service.Sales, "customers", customerId, "orders",
orderId, "orderItems",
        orderItemId);
    OrderItem orderItem = new OrderItem();
    orderItem.setQuantity(quantity);
    Entity<OrderItem> entity = Entity.entity(orderItem, MediaType.APPLICATION_JSON);
    logInfo("Executing " + webTarget.getUri() + " with " + orderItem);
    Response response = webTarget.request(MediaType.APPLICATION_JSON).build(PATCH_METHOD,
entity).invoke();
    if (isError(response)) {
        HttpErrorException exception = new HttpErrorException(response);
        logInfo("Failed to update order item: " + exception.getMessage());
        throw exception;
    } else {
        logInfo("Updated order item: " + response.readEntity(OrderItem.class));
    }
}
```

Note that the same *lookupProduct()* operation is used again here to get the product details before inventory can be checked.

The cart JSP displays the content of the customer shopping cart and allows the user to delete any order item or update its order quantity. The customer can also check out, provide a payment method and request that the order be processed. Notice that for each order item, two variables are used to display the table. The order item itself contains the quantity of the product that is ordered, but the product details from the *Product* service have been previously retrieved and are looked up from a map for each iteration:

```
<%@page
    import="com.redhat.refarch.microservices.presentation.RestClient"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<c:if test="${param.updateQuantity}">
    <%
        RestClient.updateQuantity(request);
    %>
</c:if>
<form target="_self" id="returnForm" method="post">
    <table style="width: 100%;">
        <tr>
            <c:if test="${not empty errorMessage}">
                <td>
                    <div style="color: red">${errorMessage}</div>
                </td>
            </c:if>
            <td style="float: right; border: 0; text-align: right;">
                <button name="home" id="home" value="true"
                    style="margin-right: 20px; margin-left: 20px;">Return</button>
            </td>
        </tr>
    </table>
    <c:if test="${itemCount == 0}">
        <script type="text/javascript">
            document.getElementById('returnForm').submit();
        </script>
    </c:if>
</form>
<div style="margin-top: 5em;">
    <c:forEach var="orderItem" items="${orderItems}">
        <c:set var="product" value="${inventory[orderItem.sku]}" />
        <br />
        <br />
        <table style="margin: 0px auto; width: 80%; border: 1px solid black;">
            <caption style="margin: 0px auto; font-size: 2em">${product.name}</caption>
            <tr style="border: 1px solid black;">
                <td style="border: 1px solid black; padding: 5px"></td>
            </tr>
        </table>
    </c:forEach>
</div>
```

```

        <td style="border: 1px solid black; padding:
5px">${product.description}</td>
        <td style="border: 1px solid black; padding: 5px">Product
            Dimensions: ${product.length} x ${product.width} x
            ${product.height} <br /> Product Weight: ${product.weight}
        </td>
        <td style="border: 1px solid black; padding: 5px">
            <p style="font-size: 1.5em">${product.price}</p>
            <p>Availability: ${product.availability}</p>
            <form target="_self" method="post">
                <input type="hidden" name="cart" value="true"> <input
                    type="hidden" name="orderId" value="${orderId}">
<input
                    type="number" name="quantity" size="5"
                    value="${quantity}">
                <button name="updateQuantity" id="updateQuantity" value="true"
                    type="submit">Update</button>
                <button name="delete" type="button"
                    onclick="deleteItem(this.form);">Delete</button>
            </form>
        </td>
    </tr>
</table>
</c:forEach>
</div>

<form target="_self" method="post">
    <table style="width: 100%; margin-top: 3em">
        <tr>
            <td style="text-align: center;">
                <button name="checkout" value="true"
                    style="background-color: LightBlue; font-size: 1.5em; padding:
5px;">Checkout</button>
            </td>
        </tr>
    </table>
</form>

<script type="text/javascript">
    function deleteItem(itemForm) {
        itemForm.elements["quantity"].value = 0;
        itemForm.elements["updateQuantity"].click();
    }
</script>

```

In both cases of update and delete, the form is submitted with a request to update quantity. The value for quantity is set to zero through a JavaScript function to imply a delete.

Once the form is submitted, the JSP will call the `updateQuantity` method:

```
public static void updateQuantity(HttpServletRequest request) throws HttpException {
    Customer customer = (Customer) request.getSession().getAttribute("customer");
    long customerId = customer.getId();
    Long orderId = (Long) request.getSession().getAttribute("orderId");
    Long orderItemId = Long.valueOf(request.getParameter("orderItemId"));
    int quantity = Integer.valueOf(request.getParameter("quantity"));
    if (quantity == 0) {
        deleteOrderItem(customerId, orderId, orderItemId);
    } else {
        updateOrderItem(request, customerId, orderId, orderItemId, null, quantity);
    }
    getPendingOrder(request, customerId);
}
```

In response to a zero quantity, another convenience method is called to delete the order item:

```
private static void deleteOrderItem(long customerId, long orderId, long orderItemId)
throws HttpException {
    WebTarget webTarget = getWebTarget(Service.Sales, "customers", customerId, "orders",
orderId, "orderItems",
    orderItemId);
    logInfo("Deleting " + webTarget.getUri());
    Response response = webTarget.request(MediaType.APPLICATION_JSON).delete();
    logInfo("Got response " + response.getStatus());
    if (isError(response)) {
        throw new HttpException(response);
    }
}
```

Updates to the order item quantity leverage the same convenience method previously used in response to the purchase button.

When the customer decides to check out, the checkout.jsp uses the order items stored in the session to display a table, showing the unit price, order quantity and total order prices:

```
<%@page
    import="com.redhat.refarch.microservices.presentation.RestClient"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>

<div style="margin-top: 5em; margin-bottom: 1em;">
    <table style="margin: 0px auto;">
        <caption style="margin: 0px auto; font-size: 2em; padding: 1em;">Order
            Summary</caption>
        <tr style="font-weight: bold;">
            <td style="border: 1px solid black; padding: 5px">Product</td>
            <td style="border: 1px solid black; padding: 5px">Unit Price</td>
            <td style="border: 1px solid black; padding: 5px">Quantity</td>
            <td style="border: 1px solid black; padding: 5px">Product Cost</td>
        </tr>
        <c:set var="total" value="${0}" />
        <c:forEach var="product" items="${orderItems}">
            <tr style="border: 1px solid black;">
                <td
                    style="border: 1px solid black; padding: 5px; text-align:
right;">${product.name}</td>
                <td
                    style="border: 1px solid black; padding: 5px; text-align:
right;"><fmt:formatNumber
                        value="${product.price}" type="currency" groupingUsed="true"
                    /></td>
                <td
                    style="border: 1px solid black; padding: 5px; text-align:
right;"><fmt:formatNumber
                        value="${product.quantity}" type="currency" groupingUsed="true"
                    /></td>
                <td
                    style="border: 1px solid black; padding: 5px; text-align:
right;"><fmt:formatNumber
                        value="${product.price * product.quantity}" type="currency"
                        groupingUsed="true" /></td>
            </tr>
            <c:set var="total"
                value="${total + product.price * product.quantity}" />
        </c:forEach>
        <tr style="font-weight: bold; margin-top: 1em;">
            <td style="padding: 5px;"><div style="padding-top: 1em;">Grand
                Total:</div></td>
            <td style="padding: 5px;"></td>
```

```

        <td style="padding: 5px;"></td>
        <td style="padding: 5px; text-align: right;"><div
            style="padding-top: 1em;">
            <fmt:formatNumber value="${total}" type="currency"
                groupingUsed="true" />
            </div></td>
    </tr>
</table>
</div>

```

The JSP uses the core library and a total variable to add up product prices. It also uses the formatting tag library to show prices in currency format, with grouping of every three digits.

This is followed by a credit card form to accept the customer's method of payment:

```

<form target="_self" method="post">
    <input type="hidden" name="amount" value="${total}">
    <table style="margin: 0em auto; border: 0px; padding: 2em;">
        <tr>
            <td style="padding: 5px;">Customer:</td>
            <td style="padding: 5px;">${sessionScope.customer.name}</td>
        </tr>
        <tr>
            <td style="padding: 5px;">Telephone:</td>
            <td style="padding: 5px;">${sessionScope.customer.telephone}</td>
        </tr>
        <tr>
            <td style="padding: 5px;">Address:</td>
            <td style="padding: 5px;">${sessionScope.customer.address}</td>
        </tr>
        <tr>
            <td style="padding: 5px;">Credit Card No:</td>
            <td style="padding: 5px;"><input type="text" name="creditCardNo"
                size="18" maxLength="16" pattern="\d{16}" required></td>
        </tr>
        <tr>
            <td style="padding: 5px;">Expiration Date</td>
            <td style="padding: 5px;"><select name='expirationMM'
                id='expirationMM'>
                <option value='01'>January</option>
                <option value='02'>February</option>
                <option value='03'>March</option>
                <option value='04'>April</option>
                <option value='05'>May</option>
                <option value='06'>June</option>
                <option value='07'>July</option>
                <option value='08'>August</option>
            </select>
        </td>
    </tr>
    </table>
</form>

```

```

        <option value='09'>September</option>
        <option value='10'>October</option>
        <option value='11'>November</option>
        <option value='12'>December</option>
    </select> <select name='expirationYY' id='expirationYY'>
        <option value='2015'>2015</option>
        <option value='2016'>2016</option>
        <option value='2017'>2017</option>
        <option value='2018'>2018</option>
        <option value='2019'>2019</option>
    </select></td>
</tr>
<tr>
    <td style="padding: 5px;">Verification Code</td>
    <td style="padding: 5px;"><input type="text"
        name="verificationCode" max="999" size="4" maxlength="3"
        pattern="\d{3}" required></td>
</tr>
</table>

```

Submit and cancel buttons are provided. To bypass HTML5 validation, the cancel button uses JavaScript to submit a different form:

```

<div style="margin: 0px auto; text-align: center;">
    <button name="completeOrder" value="true"
        style="background-color: LightBlue; font-size: 1em; padding: 5px; margin-
left: 20px; margin-right: 20px;">Submit</button>
    <button onclick="document.getElementById('cancel_form').submit();"
        type="button"
        style="background-color: LightBlue; font-size: 1em; padding: 5px; margin-
left: 20px; margin-right: 20px;">Cancel</button>
</div>
</form>

<form id="cancel_form" target="_self" method="post"></form>

```

The button to submit the form has the name completeOrder, which will result in a request parameter of the same name, prompting the index JSP file to call the corresponding convenience method:

```

public static void completeOrder(HttpServletRequest request) throws
    ClientProtocolException, IOException, JSONException, URISyntaxException
{
    JSONObject jsonResponse = processTransaction( request );
    String status = jsonResponse.getString( "status" );
    if( "SUCCESS".equals( status ) )

```

```
{
    @SuppressWarnings("unchecked")
    List<OrderItem> orderItems = (List<OrderItem>)request.getSession()
        .getAttribute( "orderItems" );

    try
    {
        HttpResponse response = reduceInventory( orderItems );
        if( isError( response ) )
        {
            throw new HttpErrorException( response );
        }
    }
    catch( Exception e )
    {
        refundTransaction( jsonResponse.getInt( "transactionNumber" ) );
        request.setAttribute( "errorMessage",
            "Insufficient inventory to fulfill order" );
        return;
    }
    try
    {
        markOrderPayment( request, jsonResponse );
        request.setAttribute( "successMessage",
            "Your order has been processed" );
    }
    catch( Exception e )
    {
        logInfo( "Order " + request.getSession().getAttribute( "orderId" )
            + " processed but not updated in the database" );
        request.setAttribute( "errorMessage",
            "Order processed. Allow some time for update!" );
    }
    request.getSession().removeAttribute( "orderId" );
    request.getSession().removeAttribute( "orderItems" );
    request.getSession().setAttribute( "itemCount", 0 );
}
else if( "FAILURE".equals( status ) )
{
    request.setAttribute( "errorMessage",
        "Your credit card was declined" );
}
}
```

This method calls four other convenience methods to process an order.

The *processTransaction* method is called to process the credit and receive the payment. Next, *reduceInventory* is called to try and adjust the availability of the product based on this purchase and if successful, *markOrderPayment* is called to move the items from the shopping cart to an order with a more advanced status. If inventory adjustment could not take place, either due to unforeseen errors or because the product has sold out, the *refundTransaction* method is invoked to refund the order amount to the same credit card.

To process the transaction, simply call the billing service. The response from this call will have a status indicating the success or failure of the credit card transaction, while the transaction number can be used to later cancel and refund the payment:

```
private static Result processTransaction(HttpServletRequest request) throws
HttpException {
    WebTarget webTarget = getWebTarget(Service.Billing, "process");
    Transaction transaction = Utils.getTransaction(request);
    Entity<Transaction> entity = Entity.entity(transaction, MediaType.APPLICATION_JSON);
    logInfo("Executing " + webTarget.getUri() + " with " + transaction);
    Response response = webTarget.request(MediaType.APPLICATION_JSON).post(entity);
    if (isError(response)) {
        HttpException exception = new HttpException(response);
        logInfo("Failed to process transaction: " + exception.getMessage());
        throw exception;
    } else {
        Result result = response.readEntity(Result.class);
        logInfo("Got transaction result: " + result);
        return result;
    }
}
```

To reduce the inventory based on the ordered quantities, simply use the corresponding method in the product service. This service operation uses Pessimistic Locking to ensure concurrent transactions do not result in incorrect product availability:

```
private static void reduceInventory(List<OrderItem> orderItems) throws HttpException
{
    WebTarget webTarget = getWebTarget(Service.Product, "reduce");
    List<Inventory> inventoryList = new ArrayList<>();
    for (OrderItem orderItem : orderItems) {
        Inventory inventory = new Inventory();
        inventory.setSku(orderItem.getSku());
        inventory.setQuantity(orderItem.getQuantity());
        inventoryList.add(inventory);
    }
    Entity<List<Inventory>> entity = Entity.entity(inventoryList,
MediaType.APPLICATION_JSON);
    logInfo("Executing " + webTarget.getUri() + " with " + inventoryList);
    Response response = webTarget.request(MediaType.APPLICATION_JSON).post(entity);
    if (isError(response)) {
        HttpException exception = new HttpException(response);
        logInfo("Failed to reduce inventory: " + exception.getMessage());
        throw exception;
    }
}
```

The order status is updated to Paid using a partial update to the corresponding resource in the sales service:

```
private static void markOrderPayment(HttpServletRequest request, Result
transactionResult)
    throws HttpException {
    Customer customer = (Customer) request.getSession().getAttribute("customer");
    Long orderId = transactionResult.getOrderNumber();
    WebTarget webTarget = getWebTarget(Service.Sales, "customers", customer.getId(),
"orders", orderId);
    Order order = new Order();
    order.setStatus(Status.Paid);
    order.setTransactionDate(transactionResult.getTransactionDate());
    order.setTransactionNumber(transactionResult.getTransactionNumber());
    Entity<Order> entity = Entity.entity(order, MediaType.APPLICATION_JSON);
    logInfo("Executing " + webTarget.getUri() + " with " + order);
    Response response = webTarget.request(MediaType.APPLICATION_JSON).build(PATCH_METHOD,
entity).invoke();
    if (isError(response)) {
        HttpException exception = new HttpException(response);
        logInfo("Failed to mark order payment: " + exception.getMessage());
        throw exception;
    } else {
        Order updatedOrder = response.readEntity(Order.class);
        logInfo("Order updated with payment: " + updatedOrder);
    }
}
```

In case of failure after the payment has been collected, the refund call simply calls the corresponding operation of the billing service:

```
private static void refundTransaction(long transactionNumber) throws HttpException {
    WebTarget webTarget = getWebTarget(Service.Billing, "refund", transactionNumber);
    logInfo("Executing " + webTarget.getUri());
    Response response =
webTarget.request(MediaType.APPLICATION_JSON).post(Entity.json(null));
    logInfo("Transaction refund response code: " + response.getStatus());
    if (isError(response)) {
        HttpException exception = new HttpException(response);
        logInfo("Failed to process transaction: " + exception.getMessage());
        throw exception;
    }
}
```

Finally, the *history.jsp* file displays all customer orders in response to clicking the order history link. This JSP calls a convenience method to retrieve customer orders and uses the tag library to iterate through orders and order items.

For each order, the data is displayed in an HTML table within a formatted section that includes a large margin to separate the orders:

```
%@page
    import="com.redhat.refarch.microservices.presentation.RestClient"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>

<%
    RestClient.getOrderHistory( request );
%>
<c:forEach var="order" items="${orders}">
    <div style="margin-top: 5em; margin-bottom: 1em;">
        <table style="margin: 0px auto;">
```

The caption of the table includes the order number and status. The first line of the table prints the headers for the content:

```
<caption style="margin: 0px auto; font-size: 1.5em; padding: 5px;">
    Order ${order.id}, ${order.status}</caption>
<tr style="font-weight: bold;">
    <td style="border: 1px solid black; padding: 5px;">Product</td>
    <td style="border: 1px solid black; padding: 5px;">Unit Price</td>
    <td style="border: 1px solid black; padding: 5px;">Quantity</td>
    <td style="border: 1px solid black; padding: 5px;">Product Cost</td>
</tr>
```

Each order item is displayed in one row of the table, while a *total* variable is used to keep a running total for the purchase:

```
<c:set var="total" value="${0}" />
<c:forEach var="product" items="${order.orderItems}">
  <tr style="border: 1px solid black;">
    <td style="border: 1px solid black; padding: 5px;
      text-align: right; max-width: 15em; min-width: 15em;">
      ${product.name}</td>
    <td style="border: 1px solid black; padding: 5px;
      text-align: right;">
      <fmt:formatNumber value="${product.price}"
        type="currency" groupingUsed="true" />
    </td>
    <td
      style="border: 1px solid black; padding: 5px;
        text-align: right;">
      <fmt:formatNumber value="${product.quantity}"
        type="currency" groupingUsed="true" />
    </td>
    <td
      style="border: 1px solid black; padding: 5px;
        text-align: right; min-width: 12em;">
      <fmt:formatNumber value="${product.price * product.quantity}"
        type="currency" groupingUsed="true" />
    </td>
  </tr>
  <c:set var="total" value="${total + product.price * product.quantity}"/>
</c:forEach>
```

The formatting tag library is used to display prices.

The total variable is then used to display the total price of the order:

```
<tr style="font-weight: bold; margin-top: 1em;">
  <td style="padding: 5px;"><div style="padding-top: 1em;">
    Grand Total:</div></td>
  <td style="padding: 5px;"></td>
  <td style="padding: 5px;"></td>
  <td style="padding: 5px; text-align: right;">
    <div style="padding-top: 1em;">
      <fmt:formatNumber value="${total}" type="currency"
        groupingUsed="true" />
    </div>
  </td>
</tr>
```

The transaction number and transaction date are also printed for orders that have already been processed:

```
<c:if test="${not empty order.transactionNumber}">
  <tr>
    <td style="padding: 5px;">Transaction Number:</td>
    <td style="padding: 5px;"></td>
    <td style="padding: 5px;"></td>
    <td style="padding: 5px; text-align: right;">
      ${order.transactionNumber}
    </td>
  </tr>
  <tr>
    <td style="padding: 5px;">Transaction Date:</td>
    <td style="padding: 5px;"></td>
    <td style="padding: 5px;"></td>
    <td style="padding: 5px; text-align: right;">
      <fmt:formatDate type="both" value="${order.transactionDate}" />
    </td>
  </tr>
</c:if>
```

Finally, a return button is provided in an empty form that just resets back to the home page:

```

        </table>
    </div>
</c:forEach>

<form target="_self" method="post">
    <div style="margin: 0px auto; text-align: center;">
        <button style="background-color: LightBlue; font-size: 1em;
            padding: 5px; margin-left: 20px; margin-right: 20px;">
            Return</button>
    </div>
</form>

```

The convenience method to retrieve the customer order history uses the sales service REST operation to get all orders:

```

public static void getOrderHistory(HttpServletRequest request) throws ServletException
{
    Customer customer = (Customer) request.getSession().getAttribute("customer");
    WebTarget webTarget = getWebTarget(Service.Sales, "customers", customer.getId(),
"orders");
    logInfo("Executing " + webTarget.getUri());
    Response response = webTarget.request(MediaType.APPLICATION_JSON).get();
    if (isError(response)) {
        throw new ServletException(response);
    } else {
        List<Order> orders = response.readEntity(new GenericType<List<Order>>() {
        });
        for (Order order : orders) {
            updateInventory(request, order.getOrderItems());
        }
        Collections.sort(orders, reverseOrderNumberComparator);
        request.setAttribute("orders", orders);
    }
}

```

The sales service does not have all the product details required to display the proper order history. For this purpose, the *updateInventory* is again used to fetch other product details and populate the inventory map.

Finally, the orders are sorted in reverse chronological order and set as a request attribute.

To sort orders based on reverse order number, create a simple Comparator:

```
private static Comparator<Order> reverseOrderNumberComparator =  
    new Comparator<Order>()  
{  
  
    @Override  
    public int compare(Order order1, Order order2)  
    {  
        return (int)( order2.getId() - order1.getId() );  
    }  
};
```

6. Conclusion

Microservice Architecture is an architectural style that provides a number of benefits by adopting a divide and conquer approach to software design and deployment. Microservices can be individually maintained, isolated, scaled up or down, or upgraded and replaced. The modularity of microservices can affect both the requirements and the benefits of the deployment. The best solution is not universal and entirely depends on the client environment and application requirements. After providing a thorough discussion on microservices and some of the factors that go into determining a client's needs and cost to benefit parameters, this reference architecture focuses on business-driven microservices that are not directly exposed to the outside world. An aggregation layer provides a simple and familiar interface to clients, while taking advantage of most benefits provided by this architectural style.

Appendix A: Revision History

Revision	Release Date	Author(s)
1.0	June 2016	Babak Mozaffari

Appendix B: Contributors

We would like to thank the following individuals for their time and patience as we collaborated on this process. This document would not have been possible without their many contributions.

Contributor	Title	Contribution
Mark Little	Vice President, Software Engineering	Requirements, Technical Review
Rich Sharples	Senior Director, Product Management	Requirements, Technical Review
Ken Johnson	Senior Director, Product Management	Requirements
Burr Sutter	Manager, Product Management	Technical Contributions
Bilge Ozpeynirci	Senior Product Manager - Technical	Technical Review
Rafael Benevides	Senior Product Manager - Technical	Technical Review
Edson Yanaga	Senior Product Manager - Technical	Technical Review