Abstract

The HP VAN SDN Controller is a Java-based OpenFlow controller enabling SDN solutions such as network controllers for the data center, public cloud, private cloud, and campus edge networks. This includes providing an open platform for developing experimental and special-purpose network control protocols using a built-in OpenFlow controller. This document provides detailed documentation for writing applications to run on the HP VAN SDN Controller platform.
## Contents

### 1 Introduction

- Overview .......................................................... 1
- Basic Architecture .................................................. 2
- Internal Applications vs. External Applications .................. 5
- Acronyms and Abbreviations ...................................... 6

### 2 Establishing Your Test and Development Environments .................................................. 7

- Test Environment
  - Installing HP VAN SDN Controller ............................. 7
  - Authentication Configuration .................................. 7
- Development Environment .......................................... 7
  - Prerequisites ...................................................... 7
  - HP VAN SDN Controller SDK .................................. 8

### 3 Developing Applications .......................................................... 10

- Introduction ...................................................... 10
  - Web Layer ...................................................... 12
  - Business Logic Layer ........................................... 12
  - Persistence Layer .............................................. 13
- Authentication ..................................................... 13
- REST API .......................................................... 15
  - REST API Documentation ..................................... 16
  - Rsdoc ............................................................ 16
  - Rsdoc Extension ................................................ 17
  - Rsdoc Live Reference ........................................... 17
- Audit Logging ...................................................... 19
- Alert Logging ...................................................... 20
- Configuration ..................................................... 21
- High Availability .................................................. 23
  - Role orchestration ............................................. 23
- OpenFlow ......................................................... 26
  - Message Library ............................................... 27
  - Core Controller ............................................... 33
  - Flow Rules ..................................................... 43
- Metrics Framework .................................................. 46
  - External View ................................................... 46
- GUI
  - SKI Framework - Overview ................................... 59
  - SKI Framework - Navigation Tree .............................. 60
  - SKI Framework - Hash Navigation ............................. 61
  - SKI Framework - View Life-Cycle ............................. 64
  - SKI Framework - Live Reference Application ................. 64
  - UI Extension .................................................... 65
  - Introduction .................................................... 66
  - Controller Teaming ............................................. 67
  - Distributed Coordination Service ............................. 67
  - Persistence ....................................................... 85
# 1 Introduction

This document describes the process of developing applications to run on the HP VAN SDN Controller platform.

The base SDN Controller serves as a delivery vehicle for SDN solutions. It provides a platform for developing various types of network controllers, e.g. data-center, public cloud, private cloud, campus edge networks, etc. This includes being an open platform for development of experimental and special-purpose network control protocols using a built-in OpenFlow controller.

The SDN Controller meets certain minimum scalability requirements and it provides the ability to achieve higher scaling and high-availability requirements via a scale-out teaming model. In this model, the same set of policies are applied to a region of network infrastructure by a team of such appliances, which will coordinate and divide their control responsibilities into separate partitions of the control domain for scaling, load-balancing and fail-over purposes.

## Overview

Regardless of the specific personality of the controller, the software stack consists of two major tiers. The upper Administrator tier hosts functionality related to policy deployment, management, personae interactions and external application interactions, for example slow-path, deliberating operations. The lower Controller tier, on the other hand, hosts policy enforcement, sensing, device interactions, flow interactions, for example fast-path, reflex, muscle-memory like operations. The interface(s) between the two tiers provide a design firewall and are elastic in that they can change along with the personality of the overall controller. Also, they are governed by a rule that no enforcement-related synchronous interaction will cross from the Controller to Administrator tier.

**Figure 1 Controller Tiers**
The Administration tier of the controller will host a web-layer through which software modules installed on the appliance can expose REST APIs [1] [2] (or RESTful web services) to other external entities. Similarly, modules can extend the available web-based GUI to allow network administrators and other personae to directly interact with the features of the software running on the SDN Controller.

A web application is an application that is accessed by users over a network such as the Internet or an intranet. The HP VAN SDN Controller runs on a web server as illustrated in Figure 2.

**Figure 2 Web Application Architecture**

![Web Application Architecture](image)

Servlets [3] [4] is the technology used for extending the functionality of the web server and for accessing business systems. Servlets provide a component-based, platform-independent method for building Web-based applications.

SDN applications do not implement Servlets directly but instead they implement RESTful web services [1] [2] which are based on Servlets; however RESTful web services also act as controllers as described in the pattern from Figure 3.

**Figure 3 Web Application Model View Controller Pattern**

![Web Application Model View Controller Pattern](image)

**Basic Architecture**

The principal software stack of the appliance uses OSGi framework (Equinox) [5] [6] and a container (Virgo) [7] as a basis for modular software deployment and to enforce service provider/consumer separation. The software running in the principal OSGi container can interact with other components running as other processes on the appliance. Preferably, such IPC interactions will occur using a standard off-the-shelf mechanism, for instance RabbitMQ, but they can exploit any means of IPC best suited to the external component at hand. Virgo, based on Tomcat [8], is a module-based Java application server that is designed to run enterprise Java applications with a high degree of flexibility and reliability. Figure 4 illustrates the HP VAN SDN Controller software stack.
Jersey [2] is a JAX-RS (JSR 311) reference implementation for building RESTful Web services. In Representational State Transfer (REST) architectural style, data and functionality are considered resources, and these resources are accessed using Uniform Resource Identifiers (URIs), typically links on the web. REST-style architectures conventionally consist of clients and servers and they are designed to use a stateless communication protocol, typically HTTP. Clients initiate requests to servers; servers process requests and return appropriate responses. Requests and responses are built around the transfer of representations of resources. Clients and servers exchange representations of resources using a standardized interface and protocol. These principles encourage RESTful applications to be simple, lightweight, and have high performance.

The HP VAN SDN Controller also offers a framework to develop Web User Interfaces - HP SKI. The SKI Framework provides a foundation on which developers can create a browser-based web application.

The HP VAN SDN Controller makes use of external services providing APIs that allow SDN applications to make use of them.

Keystone [9] is an external service that provides authentication and high level authorization services. It supports token-based authentication scheme which is used to secure the RESTful web services (Or REST APIs) and the web user interfaces.

Hazelcast [10] is an in-memory data grid management software that enables: Scale-out computing, resilience and fast, big data.

Apache Cassandra [10] is a high performance, extremely scalable, fault tolerant (no single point of failure), distributed post-relational database solution. Cassandra combines all the benefits of Google Bigtable and Amazon Dynamo to handle the types of database management needs that traditional RDBMS vendors cannot support.

Figure 5 illustrates with more detail the tiers that compose the HP VAN SDN Controller. It shows the principal interfaces and their roles in connecting components within each tier, the tiers to each other and the entire system to the external world.
The approach aims to achieve connectivity in a controlled manner and without creating undue dependencies on specifics of component implementations. The separate tiers are expected to interact over well-defined mutual interfaces, with decreasing coarseness from top to bottom. This means that on the way down, high-level policy communicated as part of the deployment interaction over the external APIs is broken down by the upper tier into something similar to a specific plan, which gets in turn communicated over the inter-tier API to the lower controller tier. The controller then turns this plan into detailed instructions which are either pre-emptively disseminated to the network infrastructure or are used to prime the RADIUS or OpenFlow [11] [12] controllers so that they are able to answer future switch (other network infrastructure device) queries.

Similarly, on the way up, the various data sensed by the controller from the network infrastructure, regarding its state, health and performance, gets aggregated at administrator tier. Only the administrator tier interfaces with the user or other external applications. Conversely, only the controller tier interfaces with the network infrastructure devices and other supporting controller entities, such as RADIUS, OpenFlow [11] [12], MSM controller software, and so on.
Internal Applications vs. External Applications

Internal applications ("Native" Applications / Modules) are ideal to exert relatively fine-grained, frequent and low-latency control interactions with the environment, for example, handling packet-in events. Some key points to consider when developing internal applications:

- Authored in Java or a byte-code compatible language, e.g. Scala, or Scala DSL.
- Deployed on the SDN Controller platform as collections of OSGi bundles.
- Built atop services (Java APIs) exported and advertised by the platform and by other applications.
- Export and advertise services (Java APIs) to allow interactions with other applications.
- Dynamically extend SDN Controller REST API surface.
- Dynamically extend SDN Controller GUI by adding navigation categories, items, views, and so on.
- Integrate with the SDN Controller authentication & authorization framework.
- Integrate with the SDN Controller Persistency & Distributed Coordination API.

Internal applications are deployed on the HP VAN SDN Controller and they interact with it by consuming business services (Java APIs) published by the controller in the SDK.
External applications are suitable to exert relatively coarse-grained, infrequent, and high-latency control interactions with the environment, such as path provisioning and flow inspections. External applications can have these characteristics:

- This can be written any language capable of establishing a secure HTTP connection. Example: Java, C, C++, Python, Ruby, C#, bash, and so on.
- They can be deployed on a platform of choice outside of the SDN Controller platform.
- They use REST API services exported and advertised by the platform and by other applications.
- They do not extend the Java APIs, REST APIs, or GUI of the controller.

This guide describes writing and deploying internal applications. For information about the REST APIs you can use for external applications, see the HP VAN SDN Controller REST API Reference Guide.

**Acronyms and Abbreviations**

There are many acronyms and abbreviations that are used in this document. Table 1 contains some of the more commonly used acronyms and abbreviations.

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLI</td>
<td>Command Line Interface</td>
</tr>
<tr>
<td>DTO</td>
<td>Data Transfer Object</td>
</tr>
<tr>
<td>HP</td>
<td>Hewlett-Packard</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>HTTPS</td>
<td>Hypertext Transfer Protocol Secure</td>
</tr>
<tr>
<td>HW</td>
<td>Hardware</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>OF</td>
<td>OpenFlow</td>
</tr>
<tr>
<td>OSGi</td>
<td>Open Service Gatway Initiative</td>
</tr>
<tr>
<td>OWASP</td>
<td>Open Web Application Security Project</td>
</tr>
<tr>
<td>SNMP</td>
<td>Simple Network Management Protocol</td>
</tr>
<tr>
<td>VLAN</td>
<td>Virtual LAN</td>
</tr>
</tbody>
</table>
2 Establishing Your Test and Development Environments

The suggested development environment contains two separate environments, a Test Environment and a Development Environment. It is recommended to use a different machine for each of these environments. The Test Environment is where the HP VAN SDN Controller and all the dependency systems will be installed; it will be very similar to a real deployment, however virtual machines [13] are useful during development phase. The Development Environment will be formed by the tools needed to create, build and package the application. Once the application is ready for deployment, the test environment will be used to install it.

One reason to keep these environments separated is because distributed applications may need a team set up to test the application (Cluster of controllers). Another reason is that some unit test and/or integration tests (RESTful Web Services [1] [2] for example) might open ports that are reserved for services offered or consumed by the controller.

Test Environment

Installing HP VAN SDN Controller

To install the SDN controller follow the instructions from the HP VAN SDN Controller Installation Guide [14].

Authentication Configuration

The HP VAN SDN Controller uses Keystone [9] for identity management. When it is installed, two users are created, "sdn" and "rsdoc", both with a default password of "skyline". This password can be changed using the keystone command-line interface from a shell on the system where the controller was installed: Follow the instructions from the HP VAN SDN Controller Installation Guide [14].

Development Environment

Pre-requisites

The development environment requirements are relatively minimal. They comprise of the following:

Operating System

Supported operating systems include:
- Windows 7 or later with MKS 9.4p1
- Ubuntu 10.10 or later
• OSX Snow Leopard or later.

Java

The Software Development Language used is Java SE SDK 1.6 or later. To install Java go to [15] and follow the download and installation instructions.

Maven

Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project’s build, reporting and documentation from a central piece of information [16].

To install Maven go to [16] and follow the download and installation instructions. Note that if you are behind a fire-wall, you may need to configure your ~/.m2/settings.xml appropriately to access the Internet-based Maven repositories via proxy, for more information see Maven Cannot Download Required Libraries on page 251.

Maven 3.0.4 or newer is needed. To verify the installed version of Maven execute the following command:

```bash
$ mvn -version
```

Curl

Curl (or cURL) is a command line tool for transferring data with URL syntax. This tool is optional. Follow the instruction from [17] to install Curl, or if you use Linux Ubuntu as development environment you may use the Ubuntu Software Center to install it as illustrated in Figure 6.

Figure 6 Installing Curl via Ubuntu Software Center

IDE

An IDE, or an Integrated Development Environment, is a software application that provides a programmer with many different tools useful for developing. Tools that bundled with an IDE may include: an editor, a debugger, a compiler, and more. Eclipse is a popular IDE that can be used to program in Java and for developing applications. Eclipse might be referenced in this guide.

HP VAN SDN Controller SDK

Download the HP VAN SDN Controller SDK from [18]. The SDK is contained in the *hp-sdn-sdk*.*.zip* file (for example: *hp-sdn-sdk-2.0.0.zip*). Unzip its contents in any location. To install the SDN Controller SDK jar files into the local Maven repository, execute the SDK install tool from the
directory where the SDK was unzipped, as follows (Note: Java SDK and Maven must already be installed and properly configured):

$ bin/install-sdk

To verify that the SDK has been properly installed look for the HP SDN libraries installed in the local Maven repository at:

~/.m2/repository/com/hp.

Javadoc

The controller Java APIs are documented in Javadoc format in the hp-sdn-apidoc*.jar file. Download the file and unzip its contents. To view the Java API documentation, open the index.html file. Figure 7 illustrates an example of the HP VAN SDN Controller documentation.

Figure 7 HP VAN SDN Controller Javadoc
3 Developing Applications

Internal applications (“Native” Applications / Modules) are ideal to exert relatively fine-grained, frequent and low-latency control interactions with the environment, for example, handling packet-in events. Some key points to consider when developing internal applications:

- Authored in Java or a byte-code compatible language, e.g. Scala, or Scala DSL.
- Deployed on the SDN Controller platform as collections of OSGi bundles.
- Built atop services (Java APIs) exported and advertised by the platform and by other applications.
- Export and advertise services (Java APIs) to allow interactions with other applications.
- Dynamically extend SDN Controller REST API surface.
- Dynamically extend SDN Controller GUI by adding navigation categories, items, views, and so on.
- Integrate with the SDN Controller authentication & authorization framework.
- Integrate with the SDN Controller Persistency & Distributed Coordination API.

Internal applications are deployed on the HP VAN SDN Controller and they interact with it by consuming business services (Java APIs) published by the controller in the SDK.

Introduction

Figure 8 illustrates the various classes of software modules categorized by the nature of their responsibilities and capabilities and the categories of the software layers to which they belong. Also shown are the permitted dependencies among the classes of such modules. Note the explicit separation of the implementations from interfaces (APIs). This separation principle is strictly enforced in order to maintain modularity and elasticity of the application. Also note that these represent categories, not necessarily the actual modules or components. This diagram only aims to highlight the classes of software modules.
Figure 8 HP Application Modules

Network Administrator Users

External Applications

User Interface End-Point

External Interface End-Point

Communications Implementations

Control API

Model API

Communications API

Control Implementations

Data Access API

Persistence

Business Logic

uses

implements

requires

response

request
Web Layer

Components in this layer are responsible for receiving and consuming appropriate external representations (XML, JSON, binary...) suitable for communicating with various external entities and, if applicable, for utilizing the APIs from the business logic layer to appropriately interact with the business logic services to achieve the desired tasks and/or to obtain or process the desired information.

User Interface End-Point (REST API) and end-point resources for handling inbound requests providing control and data access capabilities to the administrative GUI.

External Interface End-Point (REST API) are end-point resources for handling inbound requests providing control and data access capabilities to external applications, including other orchestration and administrative tools (for example IMC, OpenStack, etc.)

Business Logic Layer

Components in this layer fall into two fundamental categories: model control services and outbound communications services, and each of these are further subdivided into public APIs and private implementations.

The public APIs are composed of interfaces and passive POJOs [19], which provide the domain model and services, while the private implementations contain the modules that implement the various domain model and service interfaces. All interactions between different components must occur solely using the public API mechanisms.

Model API—Interfaces & objects comprising the domain model. For example: the devices, ports, network topology and related information about the discovered network environment.

Control API—Interfaces to access the modeled entities, control their life-cycles and in general to provide the basis for the product features to interact with each other.

Communications API—Interfaces which define the outbound forms of interactions to control, monitor and discover the network environment.

Control Implementations—Implementations of the control API services and domain model.

Communications Implementations—Implementations of the outbound communications API services. They are responsible for encoding / transmitting requests and receiving / decoding responses.

Health Service API—Allows an application to report its health to the controller (via the HealthMonitorable interface or proactively submitting health information to the HealthService directly via the updateHealth method) and/or listen to health events from the controller and other applications (via the HealthListener interface). There are 3 types of health statuses:

- **OK** – A healthy status to denote that an application is functioning as expected.
- **WARN** – An unhealthy status to denote that an application is not functioning as expected and needs attention. This status is usually accompanied by a reason as to why the application reports this status to provide clues to remedy the situation.
- **CRITICAL** – An unhealthy status to denote that some catastrophic event has happened to the application that affects the controller’s functionality. When the controller receives a CRITICAL event, it will assume that its functionality has been affected, and will proceed to
shutdown the Openflow port to stop processing Openflow events. If in a teaming environment, the controller will remove itself from the team.

Persistence Layer

**Data Access API**—Interfaces, which prescribe how to persist and retrieve the domain model information, such as locations, devices, topology, etc. This can also include any prescribed routing and flow control policies.

**Data Access Implementations**—Implementations of the persistence services to store and retrieve the SDN-related information in a database or other non-volatile form.

Authentication

Controller REST APIs are secured via a token-based authentication scheme. OpenStack Keystone [9] is used to provide the token-based authentication.

This security mechanism:
- Provides user authentication functionality with RBAC support.
- Completely isolates the security mechanism from the underlying REST API.
- Works with OpenStack Keystone.
- Exposes a REST API to allow any authentication server that implements this REST API to be hosted elsewhere (outside the SDN appliance).

This security mechanism does not:
- Provide authorization. Authorization needs to be provided by the application based on the authenticated subject’s roles.
- Support filtering functionality such as black-listing or rate-limiting.

To achieve isolation of security aspects from the API, authentication information is encapsulated by a token that a user receives by presenting his/her credentials to an Authentication Server. The user then uses this token (via header X-Auth-Token) in any API call that requires authentication. The token is validated by an Authentication Filter that fronts the requested API resource. Upon successful authentication, requests are forwarded to the RESTful APIs with the principal’s information such as:
- User ID
- User name
- User roles
- Expiration Date

Upon unsuccessful authentication (either no token or invalid token), it is up to the application to deny or allow access to its resource. This flexibility allows the application to implement its own authorization mechanism, such as ACL-based or even allow anonymous operations on certain resources.

The flow of token-based authentication in the HP VAN SDN Controller can be summarized as illustrated in Figure 9.
Figure 9 Token-based Authentication Flow

1) API Client presents credentials (username/password) to the AuthToken REST API.
2) Authentication is performed by the backing Authentication Server. The SDN Appliance includes a local Keystone-based Authentication Server, but the Authentication Server may also be hosted elsewhere by the customer (and maybe integrated with an enterprise directory such as LDAP for example), as long as it implements the AuthToken REST API (described elsewhere). The external Authentication Server use-case is shown by the dotted-line interactions. If the user is authenticated, the Authentication Server will return a token.
3) The token is returned back to the API client.
4) The API client includes this token in the X-Auth-Token header when making a request to the HP VAN SDN Controller’s RESTful API.
5) The token is intercepted by the Authentication Filter (Servlet Filter).
6) The Authentication Filter validates the token with the Authentication Server via another AuthToken REST API.
7) The validation status is returned back to the REST API.
8) If the validation is unsuccessful (no token or invalid token), the HP VAN SDN Controller will return a 401 (Unauthorized) status back to the caller.
9) If the validation is successful, the actual the HP VAN SDN Controller REST API will be invoked and business logics ensue.

In order to isolate services and applications from Keystone specifics, two APIs in charge of providing authentication services (AuthToken REST API’s) are published:
**Public API:**

1) Create token. This accepts username/password credentials and return back a unique token with some expiration.

**Service API:**

1) Revoke token. This revokes a given token.

2) Validate token. This validates a given token and returns back the appropriate principal’s information.

Authentication services have been split into these two APIs to limit sensitive services (Service API) to only authorized clients.

---

**REST API**

Internal applications do not make use of the HP VAN SDN Controller’s REST API, they extend it by defining their own RESTful Web Services. Internal applications make use of the business services (Java APIs) published by the controller. For external applications consult the RESTful API documentation (or Rsdoc) as described at Rsdoc Live Reference on page 17.

Representational State Transfer (REST) defines a set of architectural principles by which Web services are designed focusing on a system’s resources, including how resource states are addressed and transferred over HTTP by a wide range of clients written in different languages [20].

Concrete implementation of a REST Web service follows four basic design principles:

- Use HTTP methods explicitly.
- Be stateless.
- Expose directory structure-like URIs.
- Transfer XML, JavaScript Object Notation (JSON), or both.

One of the key characteristics of a RESTful Web service is the explicit use of HTTP. HTTP GET, for instance, is defined as a data-producing method that’s intended to be used by a client application to retrieve a resource, to fetch data from a Web server, or to execute a query with the expectation that the Web server will look for and respond with a set of matching resources [20].

REST asks developers to use HTTP methods explicitly and in a way that’s consistent with the protocol definition. This basic REST design principle establishes a one-to-one mapping between create, read, update, and delete (CRUD) operations and HTTP methods. According to this mapping:

- To create a resource on the server, use POST.
- To retrieve a resource, use GET.
- To change the state of a resource or to update it, use PUT.
- To remove or delete a resource, use DELETE.

See [1] for guidelines to design REST APIs or RESTful Web Services and Creating a REST API on page 169 for an example.
REST API Documentation

In addition to the Rsdoc, the HP VAN SDN Controller REST API provides information for interacting with the controller's REST API.

Rsdoc

Rsdoc is a semi-automated interactive RESTful API documentation. It offers a useful way to interact with REST APIs.

Figure 10 RSdoc

It is called RSdoc because it is a combination of JAX-RS annotations [2] and Javadoc [21] (Illustrated in Figure 11).
JAX-RS annotations and Javadoc are already written when implementing RESTful Web Services, and they are re-used to generate an interactive API documentation.

**Rsdoc Extension**

The HP VAN SDN Controller SDK offers a method to extend the Rsdoc to include applications specific RESTful Web Services (As the example illustrated in Figure 11). Since JAX-RS annotations and Javadoc are already written when implementing RESTful Web Services, in order to enable an application to extend the RSdoc is relatively easy and automatic: a few configuration files need to be updated. See Creating RSdoc on page 193 for an example.

**Rsdoc Live Reference**

To access the HP VAN SDN Controller’s Rsdoc (including extensions by applications):

1. Open a browser at https://SDN_CONTROLLER_ADDRESS:8443/api (As illustrated in Figure 10).
2. Get an authentication token by entering the following authentication JSON document: {
   "login":
   {
   "user": "sdn",
   "password": "skyline",
   "domain": "sdn"
   }} (as illustrated in Figure 12).

**NOTE**

Use the correct password if it was changed following instructions from Authentication Configuration on page 7.
3. Set the authentication token as the X-AUTH-TOKEN in the RSdoc and then click “Explore,” as illustrated in Figure 13. From this point all requests done via RSdoc will be authenticated as long as the token is valid.
Audit Logging

The Audit Log retains information concerning activities, operations and configuration changes that have been performed by an authorized end user. The purpose of this subsystem is to allow tracking of significant system changes. This subsystem provides an API which various components can use to record the fact that some important operation occurred, when and who triggered the operation and potentially why. The subsystem also provides means to track and retrieve the recorded information via an internal API as well as via external REST API. An audit log entry, once created, may not be modified. Audit log entries, once created, may not be selectively deleted. Audit log entries are only removed based on the age out policy defined by the administrator.

Audit Log data is maintained in persistence storage (default retention period is one year) and is presented to the end user via both the UI and the REST API layers.

The audit log framework provides a cleanup task that is executed daily (by default) that ages out audit log entries from persistent storage based on the policy set by the administrator.

An audit log entry consists of the following:

- User—a string representation of the user that performed the operation which triggered the audit log entry.
- Time-stamp—the time that the audit log entry was created. The time information is persisted in an UTC format.
- Activity—a string representation of the activity the user was doing that triggered this audit log entry.
- Data—a string description for the audit log entry. Typically, this contains the data associated with the operation.
• Origin—a string representation of the application or component that originated this audit log entry.
• Controller ID—the unique identification of the controller that originated the audit log entry.

Applications may contribute to the Audit Log via the Audit Log service. When creating an audit log entry the user, activity, origin and data must be provided. The time-stamp and controller identification is populated by the audit log framework. To contribute an audit log entry, use the

post(String user, String origin, String activity, String description)

method provided by the AuditLogService API. This method will return the object that was created. The strings associated with the user, origin and activity are restricted to a maximum of 255 characters, whereas the description string is restricted to a maximum of 4096 characters.

An example of an application consuming the Audit Log service is described at Auditing with Logs on page 215.

Alert Logging

The purpose of this subsystem is to allow for management of alert data. The subsystem comprises of an API which various components can use to generate alert data. The subsystem also provides means to track and retrieve the recorded information via an internal API as well as via external REST API. Once an alert entry has been created the state of the alert (active or not) is the only modification that is allowed.

Alert data is maintained in persistent storage (default retention period is 14 days) and is presented to the end user via both the UI and REST API layers. The alert framework provides a cleanup task that is executed daily (by default) that ages out alert data from persistent storage based on the policy set by the administrator.

An alert consists of the following:
• Severity—one of Informational, Warning or Critical
• Time-stamp—The time the alert was created. The time information is persisted in an UTC format.
• Description—a string description for the alert
• Origin—a string representation of the application or component that originated the alert
• Topic—the topic related to the alert. Users can register for notification when alerts related to a given topic or set of topics occur
• Controller ID—the unique identification of the controller that originated the alert

Applications may contribute alerts via the Alert service. When creating an alert the severity, topic, origin and data must be provided. The time-stamp and controller identification is populated by the alert framework. To contribute an alert, use the

post(Severity severity, AlertTopic topic, String origin, String data)

method provided by the AlertService API. This method returns the Alert DTO object that was created. The string associated with the origin is restricted to a maximum of 255 characters, as well as the data string.

An example of an application consuming the Alert service is described at Posting Alerts on page 212.
The SDN controller presents configurable properties and allows the end user to modify configurations via both the UI and REST API layers. The HP VAN SDN Controller uses the OSGi Configuration Admin [22] [23] and MetaType [24] [25] services to present the configuration data. For an application to provide configuration properties that are automatically presented by the SDN controller, they must provide the MetaType information for the configurable properties. The metatype information is contained in a “metatype.xml” file that must be present in the OSGi-INF/metatype folder of the application bundle.

The necessary metatype.xml can be automatically generated via the use of the Maven SCR annotations [26] and Maven SCR [27] plugin in a Maven pom.xml file for the application (See Root POM File on page 139). The SCR annotations must be included as a dependency, and the SCR plug-in is a build plugin.

Application pom.xml Example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  ...
  <dependencies>
    ...
    <dependency>
      <groupId>org.apache.felix</groupId>
      <artifactId>org.apache.felix.scr.annotations</artifactId>
      <version>1.9.4</version>
    </dependency>
  </dependencies>
</project>
```
The component can then use Annotations to define the configuration properties as illustrated in the following listing.

**Configurable Property Key Definition Example:**

```java
package com.hp.hm.impl;

import org.apache.felix.scr.annotations.*;
...
@Component (metatype=true)
public class SwitchComponent implements SwitchService {
    @Property(intValue = 100, description="Some Configuration")
    protected static final String CONFIG_KEY = "cfg.key";
    ...
}
```

The component is provided the configuration data by the OSGi framework as a Java Dictionary object, which can be referenced as a basic Map of key -> value pairs. The key will always be a Java String object, and the value will be a Java Object. A component will be provided the configuration data at component initialization via an annotated “activate” method. Live updates to a components configuration will be provided via an annotated “modified” method. Both of these annotated methods should define a Map<String, Object> as an input parameter. The following listing shows an example.

**Configurable Property Example:**

```java
... import com.hp.sdn.misc.ConfigUtils;
@Component (metatype=true)
public class SwitchComponent implements SwitchService {
    @Property(intValue = 100, description="Some Configuration")
    protected static final String CONFIG_KEY = "cfg.key";
    
    private int someCfgVariable;
    
    @Activate
    protected void activate(Map<String, Object> config) {
        someIntVariable = ConfigUtils.readInt(config, CONFIG_KEY, null, 100);
    }
    
    @Modified
    protected void modified(Map<String, Object> config) {
        
    }
}
```
someIntVariable = ConfigUtils.readInt(config, CONFIG_KEY, null, 100);
}
...
}

As the configuration property value can one of several different kinds of Java object (Integer, Long, String, etc.) a utility class is provided to read the appropriate Java object type from the configuration map. The ConfigUtils.java class provides methods to read integers, longs, strings, Booleans and ports from the configuration map of key -> value pairs. The caller must provide the following information:

- The configuration map
- The key (string) for the desired property in the configuration map
- A data Validator object (can be null)
- A default value. The default value is returned if the provided key is not found in the configuration map, if the key does not map to an Object of the desired type, or if a provided data validator object rejects the value.

A Validator is a typed class which performs custom validation on a given configuration value. For example, a data validator which only allows integer values between 10 and 20 is illustrated in the following listing.

Configurable Property Validator Example:

```java
import com.hp.sdn.misc.Validator;
public class MyValidator implements Validator<Integer> {
    @Override
    public boolean isValid(Integer value) {
        return ((10 <= value) && (value <= 20));
    }
}
```

To use this validator with the ConfigUtils class to obtain the configuration value from the configuration map, just include it in the method call:

```java
MyValidator myValidator = new MyValidator();
ConfigUtils.readInt(config, CONFIG_KEY, myValidator, 15);
```

High Availability

Role orchestration

Role Orchestration Service provides a federated mechanism to define the role of teamed controllers with respect to the network elements in the controlled domain. The role that a controller assumes in relation to a network element would determine whether it has abilities to write and modify the configurations on the network element, or has only read-only access to it.

As a preparation to exercise the Role Orchestration Service (ROS) in the HP VAN SDN Controller, there are two pre-requisite operations that needs to be carried out beforehand:
1) Create controller team: Using the teaming interfaces, a team of controllers need to be defined for leveraging High Availability features.
2) Create Region: the network devices for which the given controller has been identified as a master are grouped into “regions”. This grouping is defined in the HP VAN SDN Controller using the Region interface detailed in subsequent sections.

Once the region definition(s) are in place, the ROS would take care of ensuring that a master controller is always available to the respective network element(s) even when the configured master experiences a failure or there is effectively a disruption of the communication channel between the controller and the network device(s).

**Failover:** ROS would trigger the failover operation in two situations:

1) Controller failure: The ROS detects the failure of a controller in a team via notifications from the teaming subsystem. If the ROS determines that the failed controller instance was master to any region, it would immediately elect one of the backup (slave) controllers to assume the mastership over the affected region.

2) Device disconnect: The ROS instance in a controller would get notified of a communication failure with network device(s) via the Controller Service notifications. It would instantly federate with all ROS instances in the team to determine if the network device(s) in question are still connected to any of the backup (slave) controllers within the team. If that is the case, it would elect one of the slaves to assume mastership over the affected network device(s).

**Failback:** When the configured master recovers from a failure and joins the team again, or when the connection from the disconnected device(s) with the original master is resumed, ROS would initiate a failback operation i.e. the mastership is restored back to the configured master as defined in the region definition.

ROS exposes API’s through which interested applications can:

1) Create, delete or update a region definition
2) Determine the current master for a given device identified by a datapathId or IP address
3) Determine the slave(s) for a given device identified by a datapathId or IP address
4) Determine if the local controller is a master to a given device identified by a datapath
5) Determine the set of devices that a given controller is playing the master or slave role.
6) Register for region and role change notifications.

Details of the RegionService and RoleService APIs may be found at the Javadocs provided with the SDK. See Javadoc on page 9 for details.

**Illustrative usages of Role Service API’s**

- To determine the controller which is currently playing the role of Master to a given datapath, applications can use the following API’s depending on the specific need:

  ```java
  import com.hp.sdn.adm.role.RoleService;
  import com.hp.sdn.adm.system.SystemInformationService;
  ...
  public class SampleService {

    // Mandatory dependency.
    private final SystemInformationService sysInfoService;
  ```
private final RoleService roleService;

public void doAct() {
    IpAddress masterIp = roleService.getMaster(dpid).ip();
    if(masterIp.equals(sysInfoService.
        getSystem().getAddress())){
        log.debug("this controller is the master to {}", dpid);
        // now that we know this controller has master privilages
        // we could for example initiate write operations on the
        // datapath - like sending flow-mods
    }
}

- To determine the role that a controller is playing with respect to a given datapath

import com.hp.of.lib.msg.ControllerRole;
import com.hp.sdn.adm.role.RoleService;
import com.hp.sdn.region.ControllerNode;
import com.hp.sdn.region.ControllerNodeModel;
...
public class SampleService {
    // Mandatory dependency.
    private final RoleService roleService;
    public void doAct() {
        ...
        ControllerNode controller = new ControllerNodeModel("10.1.1.1");
        ControllerRole role = roleService.getCurrentRole(controller,deviceIp);
        switch(role){
            case MASTER:
                // the given controller has master privileges
                // we can trigger write-operations from that controller
                ... Break;
            Case SLAVE:
                // we have only read privileges
                ...
                break;
            default:
                // indicates the controller and device are not associated
                // to any region.
                break;
    }
Notification on Region and Role changes

Applications can express interest in region change notifications using the addListener(...) API in RegionService and providing an implementation of the RegionListener. A sample listener implementation is illustrated in the following listing:

Region Listener Example:

```java
import com.hp.sdn.adm.region.RegionListener;
import com.hp.sdn.adm.region.Region;
...
public class RegionListenerImpl implements RegionListener {
    ...
    @Override
    public void added(Region region) {
        log.debug("Master of new region: ", region.master());
    }

    @Override
    public void removed(Region region) {
        log.debug("Master of removed region: ", region.master());
    }
}
```

Similarly applications can express interest in role change notifications using the addListener(...) API in RoleService and providing an implementation of the RoleListener. A sample listener implementation is illustrated in the following listing:

Role Listener Example:

```java
import com.hp.sdn.adm.role.RoleEvent;
import com.hp.sdn.adm.role.RoleListener;
...
public class RoleListenerImpl implements RoleListener {
    ...
    @Override
    public void rolesAsserted(RoleEvent roleEvent) {
        log.debug("Previous master: ", roleEvent.oldMaster());
        log.debug("New master: ", roleEvent.newMaster());
        log.debug("Affected datapaths: ", roleEvent.datapaths());
    }
}
```

OpenFlow

OpenFlow messages are sent and received between the controller and the switches (datapaths) it manages. These messages are byte streams, the structure of which is documented in the OpenFlow Protocol Specification documents published by the Open Networking Foundation (ONF) [28].
The Message Library is a Java implementation of the OpenFlow specification, providing facilities for encoding and decoding OpenFlow messages from and to Java rich data types.

The controller handles the connections from OpenFlow switches and provides the means for upper layers of software to interact with those switches via the ControllerService API.

The following figure illustrates this:

**Figure 14 OpenFlow Controller**

---

**Message Library**

The Message Library is a Java implementation of the OpenFlow specification, providing facilities for encoding and decoding OpenFlow messages from and to Java rich data types.

**Design Goals**

The following are the overall design goals of the library:

- To span all OpenFlow protocol versions
  - However, actively supporting just 1.0.0 and 1.3.2
- To be extensible
  - Easily accommodating future versions
- To provide an elegant, yet simple, API for handling with OpenFlow messages
- To reduce the burden on application developers
  - Insulating developers from differences across protocol versions, as much as possible
- To expose the semantics but hide the syntax details
  - Developers will not be required to encode and decode bitmasks, calculate message lengths, insert padding, etc.
- To be robust and type-safe
  - Working with Java enumerations and types
**Design Choices**

Some specific design choices were made to establish the underlying principles of the implementation, to help meet the goals specified above.

- All OpenFlow messages are fully creatable/encodable/decodable, making the library completely symmetrical in this respect.
  - The controller (or app) never creates certain messages (such as PortStatus, FlowRemoved, MultipartReply, etc.) as these are only ever generated by the switch. Technically, we would only need to decode those messages, never encode them.
  - However, providing a complete solution allows us to emulate OpenFlow switches in Java code. This facilitates the writing of automated tests to verify switch/controller interactions in a deterministic manner.
- Message instances, for the most part, are immutable.
  - This means a single instance can be shared safely across multiple applications (and multiple threads) without synchronization.
  - This implies that the structures that make up the message (ports, instructions, actions, etc.) must also be immutable.
  - Where possible, “Data Types” will be used to encourage API type-safety – see the Javadocs for com.hp.util.ip and com.hp.of.lib.dt.
- Where bitmasks are defined in the protocol, Java enumerations are defined with a constant for each bit.
  - A specific bitmask value is represented by a Set of the appropriate enumeration constants.
  - For example: Set<PortConfig>
- A message instance is mutable only while the message is under construction (for example, an application composing a FlowMod message). To be sent through the system it must be converted to its immutable form first.
- To create and send a message, an application will:
  - Use the Message Factory to create a mutable message of the required type
  - Set the state (payload) of the message
  - Make the message immutable
  - Send the message via the ControllerService API.
- The Core Controller will use the Message Factory to encode the message into its byte-stream form, for transmitting to the switch.
- The Core Controller will use the Message Factory to decode incoming messages from their byte-stream form into their (immutable) rich data type form.
Message Composition and Type Hierarchy

All OpenFlow message instances are subclasses of the `OpenflowMessage` abstract class. Every message includes an internal `Header` instance that encapsulates:

- The protocol version
- The message type
- The message length (in bytes)
- The transaction ID (XID)

In addition to the header, specific messages may include:

- Data values, such as “port number”, “# bytes processed”, “metadata mask”, “h/w address”, etc.
  - These values are represented by Java primitives, enumeration constants, or data types.
- Other common structures, such as Ports, Matches, Instructions, Actions, etc.
  - These structure instances are all subclasses of the `OpenflowStructure` abstract class.

For each defined OpenFlow message type (see `com.hp.of.lib.msg.MessageType`) there are corresponding concrete classes representing the immutable and mutable versions of the message. For a given message type (denoted below as “Foo”) the following class relationships exist:
Each mutable subclass includes a private `Mutable` object that determines whether the instance is still “writable”. While writable, the “payload” of the mutable message can be set. Once the message has been made immutable, the mutable instance is marked as “no longer writable”; any attempt to change its state will result in an `InvalidMutableException` being thrown.

Note that messages are passive in nature as they are simply data carriers.

Note also that structures (e.g. a Match) have a very similar class relationship.

**Factories**

Messages and structures are parsed or created by factories. Since the factories are all about processing, but contain no state, the APIs consist entirely of static methods. Openflow messages are created, encoded, or parsed by the `MessageFactory` class. Supporting structures are created, encoded, or parsed by supporting factories, e.g. `MatchFactory`, `FieldFactory`, `PortFactory`, etc.

The main factory that application developers will deal with is the `MessageFactory`: 
The other factories that a developer might use are:

- **MatchFactory**—creates matches, used in FlowMods
- **FieldFactory**—creates match fields, used in Matches
- **InstructionFactory**—creates instructions for FlowMods
- **ActionFactory**—creates actions for instructions, (1.0 flowmods), and group buckets
- **PortFactory**—creates port descriptions
  - Note that there are “reserved” values (special port numbers) defined on the Port class (MAX, IN_PORT, TABLE, NORMAL, FLOOD, ALL, CONTROLLER, LOCAL, ANY)—see com.hp.of.lib.msg.Port Javadocs
- **QueueFactory**—creates queue descriptions
- **MeterBandFactory**—creates meter bands, used in MeterMod messages
- **BucketFactory**—creates buckets, used in GroupMod messages
- **TableFeatureFactory**—creates table feature descriptions

Note that application developers should not ever need to invoke “parse” or “encode” methods on any of the factories; those methods are reserved for use by the Core Controller.

**An example: creating a FlowMod message**

The following listing shows an example of how to create a flowmod message:

```java
public class SampleFlowModMessageCreation {
    private static final ProtocolVersion PV = ProtocolVersion.V_1_3;
    private static final long COOKIE = 0x00002468;
    private static final TableId TABLE_ID = TableId.valueOf(200);
```
private static final int FLOW_IDLE_TIMEOUT = 300;
private static final int FLOW_HARD_TIMEOUT = 600;
private static final int FLOW_PRIORITY = 50;
private static final Set<FlowModFlag> FLAGS = EnumSet.of(
    FlowModFlag.SEND_FLOW_REM,
    FlowModFlag.CHECK_OVERLAP,
    FlowModFlag.NO_BYTE_COUNTS
);

private static final MacAddress MAC =
    MacAddress.valueOf("00001e:000000");
private static final MacAddress MAC_MASK =
    MacAddress.valueOf("ffffff:000000");
private static final PortNumber SMTP_PORT = PortNumber.valueOf(25);

private static final MacAddress MAC_DEST = MacAddress.BROADCAST;
private static final IpAddress IP_DEST = IpAddress.LOOPBACK_IPv4;

private OfmFlowMod sampleFlowModCreation() {
    // Create a 1.3 FlowMod ADD message...
    OfmMutableFlowMod fm = (OfmMutableFlowMod)
        MessageFactory.create(PV, MessageType.FLOW_MOD,
            FlowModCommand.ADD);

    // NOTE: outPort = ANY and outGroup = ANY by default so we don’t have
    // to explicitly set them.
    // Also, bufferId defaults to BufferId.NO_BUFFER.

    fm.cookie(COOKIE).tableId(TABLE_ID).priority(FLOW_PRIORITY)
        .idleTimeout(FLOW_IDLE_TIMEOUT)
        .hardTimeout(FLOW_HARD_TIMEOUT)
        .flowModFlags(FLAGS)
        .match(createMatch());

    for (Instruction ins: createInstructions())
        fm.addInstruction(ins);

    return (OfmFlowMod) fm.toImmutable();
}

private Match createMatch() {
    // NOTE static imports of:

    // com.hp.of.lib.match.OxmBasicFieldType.*;

    MutableMatch mm = MatchFactory.createMatch(PV)
Core Controller

The Core Controller handles the connections from OpenFlow switches and provides the means for upper layers of software to interact with those switches via the ControllerService API.

Design Goals

The following are the overall design goals of the core controller:

- To support OpenFlow 1.0.0 and 1.3.2 switches.
- To provide the base platform for higher-level OpenFlow Controller functionality.
- To implement the services of:
  - Accepting and maintaining connections from OpenFlow-capable switches

Core Controller

The Core Controller handles the connections from OpenFlow switches and provides the means for upper layers of software to interact with those switches via the ControllerService API.

Design Goals

The following are the overall design goals of the core controller:

- To support OpenFlow 1.0.0 and 1.3.2 switches.
- To provide the base platform for higher-level OpenFlow Controller functionality.
- To implement the services of:
  - Accepting and maintaining connections from OpenFlow-capable switches
• Maintaining information about the state of all OpenFlow ports on connected switches
• Conforming to protocol rules for sending messages back to switches
• To provide a modular framework for controller sub-components, facilitating extensibility of the core controller.
• To provide an elegant, yet simple, API for Network Service components and SDN Applications to access the core services.
• To provide a certain degree of “sandboxing” of applications to protect them (and the controller itself) from ill-performing applications.

Design Choices

Some specific design choices were made to establish the underlying principles of the implementation, to help meet the goals specified above.

• The controller will use the OpenFlow Message Library to encode / decode OpenFlow messages; all APIs will be defined in terms of OpenFlow Java rich data-types.
• All OpenFlow messages and structures passed into and out of the controller must be immutable.
• Services and Applications may register as listeners to be notified of events such as:
  o Datapaths connecting or disconnecting
  o Messages received from datapaths
  o Packets received from datapaths (packet-in processing)
  o Flows being added to or removed from datapaths
• The controller will decouple incoming connection events and message events from the consumption of those events by listeners, using bounded event queues.
  o This will provide some level of protection for the controller and for the listeners, from an ill-performing listener implementation.
  o It is up to each listener to consume events fast enough to keep pace with the rate of arrival.
    – In the event that the listener is unable to do so, an out-of-band “queue-full” event will be posted, and event queuing for that listener will be suspended.
• Services and Applications will interact with the controller via the ControllerService API.
• The controller will be divided into several modules, each responsible for specific tasks:
  o Core Controller—listens for connections from, and maintains state information about, OpenFlow switches (datapaths).
  o Packet Sequencer—listens for Packet-In messages, orchestrates the processing and subsequent transmission of Packet-Out replies.
  o Flow Tracker—provides basic management of flow rules, meters, and groups.

Controller Service

The ControllerService API provides a common façade for consumers to interact with the controller. The implementing class (ControllerManager) delegates to the appropriate sub-component or to the core controller. The following sections briefly describe the API methods, with some code examples – see the Javadocs for more details.
In the following code examples, it is assumed that a reference to the controller service implementation has been stored in the field `cs`:

```java
private ControllerService cs = ...;
```

**Datapath Information**

Information about datapaths that have connected to the controller is available; either all connected datapaths, or a datapath with a given ID:

- `getAllDataPathInfo() : Set<DataPathInfo>`
- `getDataPathInfo(DataPathId) : DataPathInfo`

The `DataPathInfo` API provides information about a datapath:

- the datapath ID
- the negotiated protocol version
- the time at which the datapath connected to the controller
- the time at which the last message was received from the datapath
- the list of OpenFlow-enabled ports
- the reported number of buffers
- the reported number of tables
- the set of capabilities
- the remote (IP) address of the connection
- the remote (TCP) port of the connection
- a textual description
- the manufacturer
- the hardware version
- the software version
- the serial number
- a device type identifier

The following listing shows an example of how to use Datapath information:

**Datapath Information Example:**

```java
DataPathId dpid = DataPathId.valueOf("00:00:00:00:00:00:00:01");
DataPathInfo dpi;
try {
    dpi = cs.getDataPathInfo(dpid);
    log.info("Datapath with ID {} is connected", dpid);
    log.info("Negotiated protocol version is ", dpi.negotiated());
    for (Port p: dpi.ports()) {
        ...
    }
} catch (NotFoundException e) {
    log.warn("Datapath with ID {} is not connected", dpid);
}
```
Listeners

Application code may wish to be notified of events via a callback mechanism. A number of
methods allow the consumer to register as a listener for certain types of event:

- **Message Listeners** – notified when OpenFlow messages arrive from a datapath. At
  registration, the listener specifies the message types of interest. Note that one exception to
  this is PACKET_IN messages; to hear about these, one must register as a
  SequencedPacketListener.

- **Sequenced Packet Listeners** – notified when PACKET_IN messages arrive from a datapath. This
  mechanism is described in more detail in a following section.

- **Flow Listeners** – notified when FLOW_MOD messages are pushed out to datapaths, or when
  flow rules are removed from datapaths (either explicitly, or by timeout).

- **Group Listeners** – notified when GROUP_MOD messages are pushed out to datapaths.

- **Meter Listeners** – notified when METER_MOD messages are pushed out to datapaths.

The following listing shows an example that listens for `ECHO_REPLY` messages (presumably we
have some other code that is sending `ECHO_REQUEST` messages), and `PORT_STATUS` messages.

**ECHO_REPLY and PORT_STATUS Example:**

```java
private static final Set<MessageType> INTEREST = EnumSet.of(
    MessageType.ECHO_REPLY,
    MessageType.PORT_STATUS
);

private void initListener() {
    cs.addMessageListener(new MyListener(), INTEREST);
}

private class MyListener implements MessageListener {
    @Override
    public void queueEvent(QueueEvent event) {
        log.warn("Message Listener Queue event: ", event);
    }

    @Override
    public void event(MessageEvent event) {
        if (event.type() == OpenflowEventType.MESSAGE_RX) {
            OpenflowMessage msg = event.msg();
            DataPathId dpid = event.dpid();
            switch (msg.getType()) {
                case ECHO_REPLY:
                    handleEchoReply((OfmEchoReply) msg, dpid);
                    break;
                case PORT_STATUS:
                    handlePortStatus((OfmPortStatus) msg, dpid);
                    break;
            }
        }
    }
}
```


Statistics

The ControllerService API has a number of methods for retrieving various “statistics” about the controller, or about datapaths in the network.

- `getStats()`—returns statistics on byte and packet counts, from the controller’s perspective.
- `getPortStats(...)`—queries the specified datapath for statistics on its ports.
- `getFlowStats(...)`—queries the specified datapath for statistics on installed flows.
- `getGroupDescription(...)`—queries the specified datapath for its group descriptions.
- `getGroupStats(...)`—queries the specified datapath for statistics on its groups.
- `getGroupFeatures(...)`—queries the specified datapath for the group features it supports.
- `getMeterConfig(...)`—queries the specified datapath for its meter configurations.
- `getMeterStats(...)`—queries the specified datapath for statistics on its meters.
- `getMeterFeatures(...)`—queries the specified datapath for the meter features it supports.
- `getExperimenter(...)`—queries the specified datapath for meter configuration or statistics for OpenFlow 1.0 datapaths.

As an example, a method to print all the flows on a given datapath could be written as follows:

**Flows Example:**

```java
private void printFlowStats(DataPathId dpid) {
    List<MBodyFlowStats> stats = cs.getFlowStats(dpid, TableId.ALL);
    // Note: the above is a blocking call, which will wait for the
    // controller to send the request to the datapath and retrieve the
    // response, before returning.
    print("All flows installed on datapath {} ...", dpid);
    for (MBodyFlowStats fs: stats)
        printFlow(fs);
}

private void printFlow(MBodyFlowStats fs) {
    print("Table ID : {}", fs.getTableId());
    print("Duration : {} secs", fs.getDurationSec());
    print("Idle Timeout : {} secs", fs.getIdleTimeout());
    print("Hard Timeout : {} secs", fs.getHardTimeout());
}
```
print("Match : {}", fs.getMatch());
// Note: this is one area where we need to be cognizant of the version:
if (fs.getVersion() == ProtocolVersion.V_1_0)
    print("Actions : {}", fs.getActions());
else
    print("Instructions : {}", fs.getInstructions());
}

Sending Messages

Applications may construct and send messages to datapaths via the “send” methods:

- send(OpenflowMessage, DataPathId) : MessageFuture
- send(List<OpenflowMessage>, DataPathId) : List<MessageFuture>

The returned MessageFuture(s) allow the caller to choose whether to wait synchronously (block until the outcome of the request is known), or whether to do some other work and then check on the result of the request later.

When a message is sent to a datapath, the corresponding MessageFuture encapsulates the state of that request. Initially the future’s result is UNSATISFIED. Once the outcome is determined, the future is “satisfied” with one of the following results:

- SUCCESS—the request was a success; the reply message is available via reply().
- SUCCESS_NO_REPLY—the request was a success; there is no associated reply.
- OFM_ERROR—the request failed; the datapath issued an error, available via reply().
- EXCEPTION—the request failed due to an exception; available via cause().
- TIMEOUT—the request timed-out waiting for a response from the datapath.

The following listing shows a code example that attaches a timestamp payload to an ECHO_REQUEST message, then retrieves the timestamp payload from the ECHO_REPLY sent back by the datapath:

ECHO_REQUEST and ECHO_REPLY Example:

```java
private static final ProtocolVersion PV = ProtocolVersion.V_1_3;
private static final int SIZE_OF_LONG = 8;
private static final String E_ECHO_FAILED =
    "Failed to send Echo Request: {}"
private static final long REQUEST_TIMEOUT_MS = 5000;

private void latencyTest(DataPathId dpid) {
    byte[] timestamp = new byte[SIZE_OF_LONG];
    ByteUtils.setLong(timestamp, 0, System.currentTimeMillis());
    OpenflowMessage msg = createEchoRequest(timestamp);
    try {
        MessageFuture future = cs.send(msg, dpid);
        future.await(REQUEST_TIMEOUT_MS); // BLOCKS
        if (future.isSuccess()) {
            long now = System.currentTimeMillis();
            long then = retrieveTimestamp(future.reply());
```
long duration = now - then;
log.info("ECHO Latency to {} is {} ms", dpid, duration);
} else {
    log.warn(E_ECHO_FAILED, future.result());
}
} catch (Exception e) {
    log.warn(E_ECHO_FAILED, e.toString());
}

private OpenflowMessage createEchoRequest(byte[] timestamp) {
    OfmMutableEchoRequest echo = (OfmMutableEchoRequest)
        MessageFactory.create(PV, MessageType.ECHO_REQUEST);
    echo.data(timestamp);
    return echo.toImmutable();
}

private long retrieveTimestamp(OpenflowMessage reply) {
    OfmEchoReply echo = (OfmEchoReply) reply;
    return ByteUtils.getLong(echo.getData(), 0);
}

Packet Sequencer

PACKET_IN messages are handled by the controller with the Packet Sequencer module. The design of this module provides an orderly, deterministic, yet flexible, scheme for allowing code running on the controller to register for participation in the handling of PACKET_IN messages. An application wishing to participate will implement the SequencedPacketListener (SPL) interface.

The following figure illustrates the relationship between the Sequencer and the SPLs participating in the processing chain:
The Roles provide three broad bands of participation with the processing of PACKET_IN messages:

- An ADVISOR may analyze and provide additional metadata about the packet (attached as “hints” for listeners further downstream), but does not contribute directly to the formation of the PACKET_OUT message.

- A DIRECTOR may contribute to the formation of the associated PACKET_OUT message by adding actions to it; DIRECTORS may also determine that the PACKET_OUT message is ready to be sent back to the datapath, and can instruct the Sequencer to send it on its way.

- An OBSERVER passively monitors the PACKET_IN/PACKET_OUT interactions.

Within each role, SPLs are processed in order of decreasing “altitude”. The altitude is specified when the SPL registers with the controller. Between them, the role and altitude provide a deterministic ordering of the “processing chain”.

When a PACKET_IN message event occurs, the PACKET_IN is wrapped in a MessageContext which provides the context for the packet being processed. The packet is also decoded to the extent where the network protocols present in the packet are identified; this information is attached to the context.

The message context is passed from SPL to SPL (via the event() callback) in the predetermined order, but only to those SPLs where at least one of the network protocols present in the packet is also defined in the SPL’s “interest” set:

- During an ADVISOR’s event() callback, hints might be attached to the context with a call to addHint(Hint).
- During a DIRECTOR’s event() callback, the PacketOut API may be utilized to:
  - Add an action to the PACKET_OUT message under construction.
  - Clear all the actions from the PACKET_OUT message under construction.
  - Indicate to the sequencer that the packet should be blocked (i.e. not sent back to the source datapath).
o Indicate to the sequencer that the packet should be sent (i.e. the PACKET_OUT should be transmitted back to the source datapath).

- During an OBSERVER’s event callback, the context can be examined to determine the outcome of the packet processing.

- Once a DIRECTOR invokes the PacketOut.send() method from their callback, the sequencer will convert the mutable PACKET_OUT message to its immutable form and attempt to send it back to the datapath. If an error occurs during the send, this fact is recorded in the message context, and the DIRECTOR’s errorEvent() callback is invoked.

- Note that every SPL that registers with the sequencer is guaranteed to see every MessageContext (subject to their ProtocolId “interest” set).

- Here is some sample code that shows how to register as an observer of DNS packets sent to the controller in PACKET_IN messages:

```java
private static final int OBS_ALTITUDE = 25;
private static final Set<ProtocolId> OBS_INTEREST = EnumSet.of(ProtocolId.DNS);

private final MyObserver myObserver = new MyObserver();

cs.addPacketListener(myObserver, PacketListenerRole.OBSERVER,
        OBS_ALTITUDE, OBS_INTEREST);
```

```java
private static class MyObserver extends SequencedPacketAdapter {
    @Override
    public void event(MessageContext context) {
        Dns dns = context.decodedPacket().get(ProtocolId.DNS);
        reportOnDnsPacket(dns, context.srcEvent().dpid());
    }

    private void reportOnDnsPacket(Dns dns, DataPathId dpid) {
        // Since packet processing (this thread) is fast-path,
        // queue the report task onto a separate thread, then return.
        // ...
    }
}
```

- Note that event processing should happen as fast as possible, since this is key to the performance of the controller. In the example above, it is suggested that the task of reporting on the DNS packet is submitted to a queue to be processed in a separate thread, so as not to hold up the main IO-Loop.
Message Context

The `MessageContext` is the object which maintains the state of processing a `PACKET_IN` message, and the formulation of the `PACKET_OUT` message to be returned to the source datapath. When a `PACKET_IN` message is received by the controller, several things happen:

- A new `MessageContext` is created
- The `PACKET_IN` message event is attached
- The packet data (if there is any) is decoded and the Packet model attached
- A mutable `PACKET_OUT` message is created and attached (with appropriate fields set)
- The `MessageContext` is passed from listener to listener down the processing chain

The `MessageContext` provides the following methods:

- `srcEvent()` – returns the message event (immutable) containing the `PACKET_IN` message received from the datapath.
- `getVersion()` – returns the protocol version of the datapath / OpenFlow message.
- `getPacketIn()` – returns the `PACKET_IN` message from the message event.
- `decodedPacket()` – returns the network packet model (immutable) of the decoded packet data.
- `getProtocols()` – returns an ordered list of protocol IDs for the protocol layers in the decoded packet.
- `packetOut()` returns the `PacketOut` API, through which actions may be applied to the `PACKET_OUT` message under construction.
- `getCompletedPacketOut()` – returns the `PACKET_OUT` message (immutable) that was sent back to the datapath.
- `addHint(Hint)` – adds a hint to the message context.
- `getHints()` – returns the list of hints attached to the context.
- `isHandled()` – returns `true` if a DIRECTOR has already instructed the sequencer to send or block the `PACKET_OUT` message.
- `isBlocked()` – returns `true` if a DIRECTOR has already instructed the sequencer to block the `PACKET_OUT` message.
- `isSent()` – returns `true` if a DIRECTOR has already instructed the sequencer to send the `PACKET_OUT` message.
- `isTestPacket()` – returns `true` if the associated packet has been determined to be a diagnostic test packet.
- `requiresProcessing()` – returns `true` if the associated packet is not a test packet, and has not yet been blocked or sent.
- `failedToSend()` – returns `true` if the attempt to send the `PACKET_OUT` message failed.
- `toDebugString()` – returns a detailed, multi-line string representation of the message context.
**Flow Tracker and Pipeline Manager**

The Flow Tracker is a sub-component of the core controller that facilitates management of flow rules, meters and groups across all datapaths managed by the controller. Its functionality is accessed through the `ControllerService` API.

The Pipeline Manager is a sub-component that maintains an in-memory model of the flow table capabilities of (1.3) datapaths. When an application attempts to install a flow, the flow tracker will consult the pipeline manager to choose a suitable table in which to install the flow, if no explicit table ID has been provided by the caller.

**Flow Management**

Flow management includes:

- Getting flow statistics from a specified datapath, for one or all flow tables
- Adding or modifying flows on a specified datapath
- Deleting flows from a specified datapath

See the earlier Message Library section for an example of how to create a `FLOW_MOD` message.

**Group Management**

Group management includes:

- Getting group descriptions from a datapath, for one or all groups.
- Getting groups statistics from a datapath, for one or all groups.
- Sending group configuration to a datapath.

Note that groups are only supported for OpenFlow 1.3 datapaths.

**Meter Management**

Meter management includes:

- Getting meter configurations from a datapath, for one or all meters
- Getting meter statistics from a datapath, for one or all meters.
- Sending meter configuration to a datapath

Note that meters are only supported for OpenFlow 1.3 datapaths. However, some 1.0 datapaths can support metering through the use of `EXPERIMENTER` messages.

**Flow Rules**

The primary mechanism used in the implementation of SDN applications is the installation of flow rules (aka “FlowMods”) on datapaths (aka switches).

**Flow Classes**

Before a FlowMod can be constructed and sent via the controller service, a corresponding “Flow Class” must be registered. The flow class explicitly defines the match fields that will be present in the flow, and the types of actions that will be taken when the flow rule is matched. The registration of
flow classes also enables the controller to arbitrate flow priorities and therefore minimize conflicts amongst co-resident SDN applications.

A flow class can be registered with code similar to the following:

```java
import static com.hp.of.ctl.prio.FlowClass.ActionClass.FORWARD;
import static com.hp.of.lib.match.OxmBasicFieldType.*;

private static final String L2_PATH_FWD = "com.foo.app.l2.path";
prefix static final String PASSWORD = "aPjk57";
promise static final String L2_DESC = "Reactive path forwarding flows";

private volatile ControllerService controller = ...; // injected reference
private FlowClass l2Class;

private void init() {
    l2Class = new FlowClassRegistrator(L2_PATH_FWD, PASSWORD, L2_DESC)
        .fields(ETH_SRC, ETH_DST, ETH_TYPE, IN_PORT)
        .actions(FORWARD).register(controller);
}
```

On creating the Registrator, the first parameter is a logical name for the flow class, the second parameter is a password used to verify ownership of the flow class (typically via the REST API), and the third parameter is a short text description of the class (that is displayed in the UI).

“fields” should specify the list of match fields that will be set in the match; “actions” is the class of actions that will be employed in the actions/instructions of the FlowMod.

Note the use of static imports making the code more concise and easier to read.

The flow class instance created by the controller service is needed to inject both the controller-assigned priority and controller-assigned base cookie for the class. On creating the flow mod message, code such as the following might be used:

```java
private static final long MY_COOKIE = 0x00beef00;
promise static final ProtocolVersion pv = ProtocolVersion.V_1_3;

OfmMutableFlowMod flow = (OfmMutableFlowMod) MessageFactory.create(pv,
    MessageType.FLOW_MOD, FlowModCommand.ADD);
flow.cookie(l2Class.baseCookie() | MY_COOKIE)
    .priority(l2Class.priority());
// ... set match fields and actions ...
// ... send flow ...
```
The flow class is assigned a unique “base cookie” (top 16 bits of the 64 bit field) which must be "OR"ed with any cookie value that you wish to include in the flow (bottom 48 bits of the 64 bit field).

The flow class “priority” is a private, logical key to be stored in the FlowMod “priority” field. It is used by the controller to look up the pre-registered flow class record, so that the match fields and actions of the FlowMod can be validated against the list of intended matches/actions.

When your application gets uninstalled, be sure to unregister any flow classes you created:

```java
private void cleanup() {
    controller.unregisterFlowClass(l2Class, PASSWORD);
}
```

**Flow Contributors**

When a datapath first connects to the controller, an initial handshaking sequence is employed. In brief...

1. Datapath connects
2. OpenFlow handshake (Hello/Hello, FeaturesRequest/Reply)
3. Extended handshake (MP-Request: Description, Ports, TableFeatures)
4. Device type determined
5. “Delete ALL Flows” command sent to datapath
6. Core “initial flows” generated
7. Contributed “initial flows” collated
8. Flows validated (via pre-registered flow classes)
9. Flows adjusted (via device driver subsystem)
10. Flows (and barrier request) sent to datapath
11. DATAPATH_READY event emitted

A component may implement `InitialFlowContributor` and register itself with the controller service. During step (7) above, the `provideInitialFlows(...)` callback method will be invoked on every registered contributor, requesting any flows to be included in the set of initial flows to be laid down on the newly-connected datapath.

A possible implementation might look like this:

```java
@override
public List<OfmFlowMod> provideInitialFlows(DataPathInfo info, boolean isHybrid) {
    List<OfmFlowMod> result = new ArrayList<>();
    if (isHybrid)
        result.add(buildFlowMod(info));
    return result;
}
```
Note that the `info` parameter provides information about the newly-connected datapath, and the `isHybrid` parameter indicates whether the controller is configured for hybrid mode or not.

Such a component must register with the controller service to have its callback invoked at the appropriate times:

```java
controller.registerInitialFlowContributor(this);
```

**Metrics Framework**

The fundamental objectives to be addressed by the metering framework are as follows.

- Support components that are part of the HP VAN SDN Controller Framework and applications that are not.
- Make metrics simple to use.
- Support the creation and updating of metrics within the controller and from outside, to accommodate apps that have external components but want to keep all of their metric data in one repository.
- Support several metric types:
  - Counter
  - Gauge
  - Rolling counter
  - Ratio gauge
  - Histogram
  - Meter
  - Timer
- Designed to be robust
  - Maintains functionality when the controller stops and restarts
  - Maintains functionality when the metering framework stops and restarts, but the controller does not
- Support persistence of data over time on different time scales.
- Support display of specified metrics via JMX.
- Support authorization-based REST access to persisted data over time.

**External View**

The overarching purpose of metering support is to provide a centralized facility that application developers can use to track metric values over time, and to provide access to the resulting time stamped values thereafter via REST. The use of this facility, as shown in the following conceptual
diagram, should demand relatively little effort from a developer beyond creating and updating the metrics they wish to utilize.

**Figure 19 Metrics Architecture**

Essentially a component or application must contact the `MetricService` to create a new `TimeStampedMetric` on their behalf; they will be returned a reference to the resulting (new) `TimeStampedMetric` object. The developer can then manipulate the returned `TimeStampedMetric` object as appropriate for their own needs, updating its value at their own cadence, on a regular or irregular basis, to reflect changes in whatever is being measured.

Behind the scenes, the `MetricService` API is backed by a `MetricManagerComponent` OSGi component. This component delegates almost all of its work to a `MetricManager` singleton, which (conceptually) contains a centralized Collection of the `TimeStampedMetric` references doled out at the request of other components and applications. This Collection of `TimeStampedMetric` references allows the metering framework to process the `TimeStampedMetric` en masse, irrespective of which application or component requested them, in a fashion that is completely decoupled from the requesting application’s or component’s use of the `TimeStampedMetric`.

The most essential processing done by the metering framework is to periodically persist `TimeStampedMetric` values to disk, and to expose “live” `TimeStampedMetric` values through JMX. Other processing is also done, such as aging out old `TimeStampedMetric` values. Decoupled from this ongoing persistence of `TimeStampedMetric` values that are still being used, values that have already been persisted from `TimeStampedMetric` over time may be read via the REST API and exported for further analysis or processing outside the controller.

**TimeStampedMetric Types**

There are seven types of `TimeStampedMetric`. They are listed below, with an example of how each type might be used.

- **TimeStampedCounter**
  - A cumulative measurement that is incremented or decremented when some event occurs.
  - Example application: the number of OpenFlow devices discovered by the controller.
- **TimeStampedGauge**
An instantaneous measure.
Example application: the amount of disk space consumed by metric data.

- **TimeStampedHistogram**
  - A distribution of values from a stream of data for which mean, minimum, maximum, and various quartile values are tracked.
  - Example application: distribution of OpenFlow flow sizes.

- **TimeStampedMeter**
  - Aggregates event durations to measure event throughput.
  - Example application: the frequency with which OpenFlow flow requests are sent to the controller by a specific switch.

- **TimeStampedRatioGauge**
  - A ratio between two non-cumulative instantaneous numbers.
  - Example application: the amount of disk space consumed by a specific application’s metric data compared to all metric data.

- **TimeStampedRollingCounter**
  - A cumulative measurement that is asymptotically increased when some event occurs, and may eventually roll over to zero and begin anew.
  - Example application: a MIB counter that represents the number of octets observed in a specific subnet.

- **TimeStampedTimer** (combines the functionality of TimeStampedHistogram and TimeStampedMeter)
  - Aggregates event durations to provide statistics about the event duration and throughput.
  - Example application: the rate at which entries are placed on a queue and a histogram of the time they spent on the queue.

**TimeStampedMetric Life Cycle**

**Creating a TimeStampedMetric**

It is possible to create a TimeStampedMetric and track its value from a component or application that is running within the controller.

To request that the MetricService create a new TimeStampedMetric, a component or application must provide a MetricDescriptor object that specifies the characteristics of the desired TimeStampedMetric. A MetricDescriptor contains four fields that, when combined, produce a combination (four-tuple) that is unique to that MetricDescriptor and the resulting TimeStampedMetric: an application ID, a primary tag, a secondary tag, and a metric name. The MetricDescriptor also contains other fields, as follows.

**Required Field(s)**

- A name that is unique among TimeStampedMetrics of the same application ID, primary tag, and secondary tag combination (String).

**Optional Field(s)**

- The ID of the application creating the TimeStampedMetric instance (String, defaulted to the application ID).
- A primary tag (String, no default).
- A secondary tag (String, no default).
- A description (String, no default).
- The summary interval in minutes (enumerated value, defaulted to 1 minute).
- Whether values for the resulting TimeStampedMetric should be visible to the controller’s JMX server (boolean, defaulted to false).
- Whether values for the resulting TimeStampedMetric should be persisted (boolean, defaulted to true).

The summary interval uses an enumerated data type to restrict the possible values to 1, 5, or 15 minutes. Also, note that while the value of most TimeStampedMetrics will likely be persisted over time there may be cases, for example troubleshooting metrics, in which it is not desired to persist the values as a time series but just to view them in real time via JMX.

The primary and secondary tags are provided as a means of grouping metrics for a specific application. For example, consider an application that is to monitor router port statistics; it might have collected a metric called TxFrames from every port of every router. The primary and secondary tags would then be used to segment the occurrences of the TxFrames metric from each router port. For some router A, port X, the four-tuple that identifies the specific instance of TimeStampedMetric corresponding to that port might be as follows.

- Application ID—com.acme.app
- Primary tag—RouterA
- Secondary tag—PortX
- Metric name—TxFrames

There is a MetricDescriptor subclass that corresponds to each type of TimeStampedMetric. These MetricDescriptor subtypes can only be created by using the corresponding MetricDescriptorBuilder subclasses. The relationship between the desired TimeStampedMetric type, corresponding MetricDescriptor subtype, and the MetricDescriptorBuilder subclasses to use to produce an instance of the right MetricDescriptor subtype are summarized below.

<table>
<thead>
<tr>
<th>TimeStampedMetric Subtype</th>
<th>Corresponding MetricDescriptor Subtype</th>
<th>Required MetricDescriptorBuilder Subtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimeStampedCounter</td>
<td>CounterDescriptor</td>
<td>CounterDescriptorBuilder</td>
</tr>
<tr>
<td>TimeStampedGauge</td>
<td>GaugeDescriptor</td>
<td>GaugeDescriptorBuilder</td>
</tr>
<tr>
<td>TimeStampedHistogram</td>
<td>HistogramDescriptor</td>
<td>HistogramDescriptorBuilder</td>
</tr>
<tr>
<td>TimeStampedMeter</td>
<td>MeterDescriptor</td>
<td>MeterDescriptorBuilder</td>
</tr>
<tr>
<td>TimeStampedRatioGauge</td>
<td>RatioGaugeDescriptor</td>
<td>RatioGaugeDescriptorBuilder</td>
</tr>
</tbody>
</table>
Using MetricDescriptorBuilders represents the application of a well-known design pattern that allows most of the fields of each MetricDescriptor subtype instance that is produced to be defaulted to commonly-used values. Thus, for a typical use case in which the defaults are applicable, the component or application that is using a MetricDescriptorBuilder to produce a MetricDescriptor subtype instance can specify values only for the fields of the MetricDescriptorBuilder subtype that are to differ from the default values.

**Call MetricService**

Once a MetricDescriptor has been created, the component or application creating a TimeStampedMetric can invoke the appropriate MetricService method for the metric type they wish to create. The MetricService methods that pertain to TimeStampedMetric creation are listed below. Note that the creation of one TimeStampedMetric type, TimeStampedRollingCounter, offers the option to specify an extra parameter above and beyond the properties conveyed by the MetricDescriptor object.

**MetricService:**

```java
public interface MetricService {
    public TimeStampedCounter createCounter(CounterDescriptor descriptor);
    public TimeStampedGauge createGauge(GaugeDescriptor descriptor);
    public TimeStampedHistogram createHistogram(HistogramDescriptor descriptor);
    public TimeStampedMeter createMeter(MeterDescriptor descriptor);
    public TimeStampedRatioGauge createRatioGauge(RatioGaugeDescriptor descriptor);
    public TimeStampedRollingCounter createRollingCounter(RollingCounterDescriptor descriptor);
    public TimeStampedRollingCounter createRollingCounter(RollingCounterDescriptor descriptor, long primingValue);
    public TimeStampedTimer createTimer(TimerDescriptor descriptor);
}
```

The optional extra parameter for the TimeStampedRollingCounter is an initial priming value for the rolling counter that will be used to take subsequent delta values. Otherwise the value of the TimeStampedRollingCounter instance the first time it should be persisted will instead be used to prime the rolling counter and no value will be observed until its second persistence occurs.

Upon acquiring a TimeStampedMetric instance from the MetricService, the component or application that requested the creation has a reference to the resulting TimeStampedMetric. The value of the TimeStampedMetric may be updated whenever the component or application wishes, as frequently or infrequently as desired, on a schedule or completely asynchronously; the framework’s interaction with the TimeStampedMetric is unaffected by these factors. The method(s) that may be used to update the value of a TimeStampedMetric will depend upon the type of TimeStampedMetric. Each time the value of a TimeStampedMetric is updated, a time stamp in the
TimeStampedMetric is updated, relative to the controller’s system clock, to indicate when the update occurred; this time stamp is used by the framework in processing the resultant values.

The following methods may be used to update the value of each TimeStampedMetric type.

- **TimeStampedCounter**
  - `dec()`—Decrements the current count by one.
  - `dec(long)`—Decrements the current count by the specified number.
  - `inc()`—Increments the current count by one.
  - `inc(long)`—Increments the current count by the specified number.

- **TimeStampedGauge**
  - `setValue(long)`—Stores the latest snapshot of the gauge value.

- **TimeStampedHistogram**
  - `update(int)`—Adds the specified value to the sample set stored by the histogram.
  - `update(long)`—Adds the specified value to the sample set stored by the histogram.

- **TimeStampedMeter**
  - `mark()`—Marks the occurrence of one event.
  - `mark(long)`—Marks the occurrence of the specified number of events.

- **TimeStampedRatioGauge**
  - `updateNumerator(double)`—Stores the latest snapshot of the numerator value.
  - `updateDenominator(double)`—Stores the latest snapshot of the denominator value.
  - `update(double, double)`—Stores the latest snapshot of both numerator and denominator values.

- **TimeStampedRollingCounter**
  - `setLatestSnapshot(long)`—Stores the latest snapshot of the rolling counter.

- **TimeStampedTimer**
  - `time(Callable<T>)`—Measures the duration of execution for the provided Callable and incorporates it into duration and throughput statistics.
  - `update(int)`—Adds an externally-recorded duration in milliseconds.
  - `update(long)`—Adds an externally-recorded duration in milliseconds.

### Unregistering a TimeStampedMetric

Depending upon where its creation was initiated, from within or from outside the controller, the collection of values from a TimeStampedMetric may be halted by a component or an application that is running within the controller or from outside of the controller via the southbound metering REST interface.

When the component or application that requested the creation of a TimeStampedMetric wishes to stop the metering framework from processing a TimeStampedMetric, presumably in preparation for destroying it, it must do so via the following MetricService method.

**Metric Removal API:**
public interface MetricService {
    public void removeMetric(TimeStampedMetric toRemove);
}

This method effectively unregisters the TimeStampedMetric from the metering framework so that the framework no longer holds any references to it and thus no longer exposes it via JMX, summarizes and persists its values, or does any other sort of processing on the TimeStampedMetric. Whether the TimeStampedMetric is subsequently destroyed by the component or application that requested its creation, it has disappeared from the framework’s viewpoint.

**Reregistering a TimeStampedMetric**

If the controller bounces (goes down and then comes back up), all components and applications that are using TimeStampedMetrics within the controller will be impacted as will the metering framework; presumably they will initialize themselves in a predictable fashion, and if they register their TimeStampedMetrics following the bounce using the same MetricDescriptor information they used before the bounce metering should recover fine; the same UIDs will be assigned to their various TimeStampedMetrics that were assigned before the bounce and the net effect will be a gap in the data on disk for TimeStampedMetrics whose values are persisted. But for application components outside the controller that created and are updating TimeStampedMetrics there may be no indication that the controller has bounced - or gone down and stayed down - until the next time they try to update TimeStampedMetric values.

Another possible, albeit unlikely, failure scenario arises should the metering service bounce while other components and applications do not; this could happen if someone killed and restarted the metering OSGi bundle. If this occurred, any components or applications that are using TimeStampedMetrics within the controller might be oblivious to the bounce as their references to the TimeStampedMetrics they requested will still be present, but they will be effectively unregistered from the metering framework when it reinitializes. The UIDs and MetricDescriptor data will be preserved by the framework for TimeStampedMetrics that have their data persisted, but they will appear to be TimeStampedMetrics that are no longer in use and just have persisted data that is waiting to be aged out. Again, for application components outside the controller that created and are updating TimeStampedMetrics there may be no indication that the metering service has bounced until the next time they try to update TimeStampedMetric values.

In order to be notified that the MetricService has gone down and/or come up, the OSGi component that corresponds to a component or application using TimeStampedMetrics should bind to the MetricService; then a method will be invoked when either occurrence happens to the MetricService and the component or application can react accordingly. There is no change to normal TimeStampedMetric creation required to handle the first failure scenario outlined above, as all OSGi components within the controller will recover after a bounce just as they do whenever the controller is initialized. But for the second failure scenario above, there is a way that a component or application can react when notified that the metering service has initialized following a bounce in which the component or application that owns TimeStampedMetrics has not bounced.

To handle such a scenario, components or applications should keep a Collection of the TimeStampedMetrics that they allocate; each TimeStampedMetric that is created on their behalf should be added to the Collection. When the entire controller is initializing and the component or application is notified that the MetricService is available this Collection will be empty or perhaps not even exist yet, but in the second failure scenario above the Collection should contain references to the pertinent TimeStampedMetrics when the MetricService becomes available. The
component or application can then iterate through the Collection, calling the following MetricService method for each TimeStampedMetric.

Metric Registration API:

```java
public interface MetricService {
    public void registerMetric(TimeStampedMetric toRegister);
}
```

This will re-register the existing TimeStampedMetric reference with the metering framework. Depending upon how long the bounce took there may be a gap in the resulting data on disk for TimeStampedMetrics that are to be persisted. It is also possible, depending on the type of TimeStampedMetric, that the value produced by the first interval summary following the bounce is affected by the bounce. For example, since TimeStampedRollingCounters take the delta of the last value reported and the previous value reported, there could be a spike in value that spans the entire time of the bounce in the first value persisted for a TimeStampedRollingCounter.

### Time Series Data

As noted for the preceding northbound REST API for data retrieval, time series values returned from the REST API for TimeStampedMetrics may be returned in "raw" form or may be further summarized to span specified time intervals. In "raw" form TimeStampedMetric values will be returned at the finest granularity possible; if the values for the TimeStampedMetric specified were summarized and persisted every minute then "raw" data will be returned such that each value spans a one-minute interval, and if the values for a particular Metric were summarized and persisted every five minutes then "raw" data will be returned such that each value spans a five-minute interval. If time series data is requested for a TimeStampedMetric at a granularity that is finer than that with which the TimeStampedMetric values were persisted, for example data is requested at one-minute intervals for a TimeStampedMetric whose values were persisted every fifteen minutes, an error will be returned to alert to the user that their request cannot be fulfilled.

It is important to note that while the persisted time series data for a given corresponding TimeStampedMetric is computed from values that the TimeStampedMetric is updated with, the resulting persisted data will typically not have the same form as the values that the TimeStampedMetric is updated with. For example, consider the case of the TimeStampedRollingCounter metric type; while TimeStampedRollingCounters are updated with 64-bit rolling counter values, the only value persisted for such a metric is the delta between two such 64-bit values (the 64-bit values themselves are not persisted). Generally speaking, the value persisted for a TimeStampedMetric is the change in its value since the last time the TimeStampedMetric’s value was persisted. This approach focuses the resulting data on what each TimeStampedMetric was measuring during a persistence interval, rather than the mechanics used to convey the measurements.

### Returned Data

The content returned for each data point, whether "raw" or summarized, differs somewhat depending upon the type of TimeStampedMetric the data resulted from. For "raw" data this content is essentially just a JSON representation of the data persisted for each data point being retrieved. For summarized data values that are computed from "raw" values, the content takes the same form as that of a "raw" data point except that the values represent the combination of all "raw" data points from the summarized interval. The content provided for each data point includes the following.
• When the value of the TimeStampedMetric that the data point was formulated from was last updated
• How many milliseconds (prior to the last update time) are encompassed by the reported value
• The value measured over the milliseconds spanned by the data point
• Sufficient information is thus provided should the data recipient wish to normalize the data to a standard interval length to smooth fluctuations in value that may be introduced by variations in the milliseconds spanned by time series values.

Summarized Values

Time series values may also be requested from the REST API in a form that is not "raw", such that each value returned represents a longer interval than the "raw" values persisted for a TimeStampedMetric. In this case the necessary data must be read in "raw" form from the data store and further summarized to produce values that span the requested interval before being returned. For example, if a particular TimeStampedMetric's values were persisted every five minutes and the REST API was invoked to retrieve hourly time series values for that TimeStampedMetric, twelve "raw" values that each span five minutes would be read from the data store and combined to produce a single resulting data point that spans the same hour encompassed by the twelve "raw" data points.

There may be gaps in the "raw" data points that span a specific interval when summarized values are returned. Continuing the preceding example of returning values that each represent an hour interval with "raw" data points that each represent five minutes, one would typically expect that twelve such "raw" data values would be summarized to produce one returned value. But in some cases there could be gaps in the "raw" data for a given hour, for example for one hour span there may be only ten "raw" data points persisted. Such gaps should be relatively infrequent and may be caused by various situations; the source of the metric’s data, perhaps a device on the network, might be inaccessible, or perhaps the controller rebooted. The effect of any such gaps will be accounted for in the summarized values that are returned; the information provided by each resulting value is sufficient for the recipient to normalize the data to smooth any inconsistencies introduced by gaps if so desired.

When summarized values are returned each resulting value represents the summary of a set of "raw" data points. These sets must be anchored somehow in the total time span encompassed by the REST request. For example, the time series data requested could be for a week of hourly data ending at the current time. Suppose that the "raw" data points for the specified metric were persisted at one-minute intervals, but that they started only four days ago; the first hour of data returned will span a time interval that starts at the time of the oldest data point within the time span encompassed by the REST request, in this case beginning four days ago. Each summarized value will be produced from "raw" data points that are offset from the starting time of the first data point returned. Continuing our example every hour value returned will be produced by "raw" minute data points that are offset by some multiple of 60 minutes from starting time of the first returned data point, four days ago in this case.

The technique used to summarize "raw" TimeStampedMetric values to produce summarized values is contingent upon the type of TimeStampedMetric the data resulted from. For all TimeStampedMetric types, the milliseconds spanned by each "raw" value are simply summed over the specified interval and the latest update time stamp among the "raw" values is reported as the last updated time stamp of the resulting value.
• **TimeStampedCounter**
  - Counts from each "raw" data point are summed, producing a long value for the total count during the summarized interval.

• **TimeStampedGauge**
  - Values from each "raw" data point are averaged, producing a double value for the gauge reading during the summarized interval.

• **TimeStampedHistogram**
  - Sample counts from the "raw" data points are summed and the minimum and maximum for the interval are computed by finding the lowest minimum and highest maximum among the "raw" data points, producing three long values for the total sample count and minimum and maximum sample values during the summarized interval. The means of the "raw" data points are averaged and their standard deviations combined, producing two double values for the mean and standard deviation of the sample values during the summarized interval.

• **TimeStampedMeter**
  - Sample counts from the "raw" data points are summed and rates from the "raw" data points are averaged, producing a long value for the total sample count and a double value for the average rate during the summarized interval.

• **TimeStampedRatioGauge**
  - Ratio values from each "raw" data point are averaged, producing double values for the numerator and denominator readings during the summarized interval.

• **TimeStampedRollingCounter**
  - Delta values from each "raw" data point are summed, producing a long value for the total delta during the summarized interval.

• **TimeStampedTimer**
  - Sample counts from the "raw" data points are summed and the minimum and maximum for the interval are computed by finding the lowest minimum and highest maximum among the "raw" data points, producing three long values for the total sample count and minimum and maximum sample values during the summarized interval. The means and rates of the "raw" data points are averaged and their standard deviations combined, producing three double values for the mean, average rate, and standard deviation of the sample values during the summarized interval.

**JMX Clients**

JConsole or another JMX client may be used to connect to the HP VAN SDN Controller’s JMX server to view selected metric values "live". Access is only permitted for local JMX clients, so any such clients must be installed on the controller system. No JMX clients are delivered with the controller or are among the prerequisites for installing it; they must be installed separately. For example, the openjdk-7-jdk package must be installed on the controller system to use JConsole.

Which TimeStampedMetrics are exposed via JMX is determined at the time of their creation, by a field in the MetricDescriptor used to create each TimeStampedMetric. Once the controller has been properly configured to permit local JMX access the user can inspect the exposed TimeStampedMetrics as they are updated "live" by the components or applications within the controller or external application components that created them.
The content exposed for each TimeStampedMetric is contingent on the type of TimeStampedMetric, but generally speaking the “live" values used by the TimeStampedMetric are visible as they are updated by the creator of the TimeStampedMetric. Using JConsole as an example, one will see a screen somewhat like Figure 20 (the exact appearance will depend upon what JVMs are running on the system):

**Figure 20 JConsole – New Connection**

Choose a local connection to the JMX server instance that looks like the one highlighted in the preceding screenshot and click the Connect button. Upon successfully connecting to that JMX server instance, one should see a screen that looks something like Figure 21.
In the list of nodes shows on the left, note the one that says HP VAN SDN Controller; this is the node under which all metrics exposed via JMX will be nested. Each application installed on the HP VAN SDN Controller will have a similar node under which all of the metrics exposed by that application are nested. Expanding the node will reveal all of the exposed metrics, which will look something like Figure 22 (note that this is just an example; real metrics will have different names).
The name displayed for each TimeStampedMetric is a combination of the primary and secondary tags and metric name specified in its MetricDescriptor during its creation; this combination will be unique among all TimeStampedMetrics monitored for a specific application. If the optional primary and/or secondary tags are not specified then only the fields provided will be used to formulate the displayed name for the TimeStampedMetric. One may select a listed metric to expand the node on the left. Selecting the Attributes subnode displays properties of the TimeStampedMetric that are exposed via JMX.
The metric UID, value field(s), and time spanned by the reported value (in seconds) are among the attributes that will be displayed.

For those TimeStampedMetrics that are persisted as well as exposed via JMX, it is possible to see the seconds get reset when the value is stored; otherwise they grow forever.

**GUI**

**SKI Framework - Overview**

The SKI Framework provides a foundation on which developers can create a browser-based web application. It is a toolkit providing assets that developers can use to construct a web-based Graphical User Interface, as shown in Figure 24.

- Third Party Libraries: (Client Side):
  - jQuery—A popular, powerful, general purpose, cross-browser DOM manipulation engine
  - jQuery UI—An extension to jQuery, providing UI elements (widgets, controls, …)
  - jQuery UI layout—An extension to jQuery, providing dynamic layout functionality
  - SlickGrid—grid/table implementation
• SKI Assets (Client Side):
  o HTML Templates—providing alternate layouts for the UI
  o Core SKI Framework—providing navigation, search, and basic view functionality
  o Reference Documentation—documenting the core framework and library APIs
  o Reference Implementation—providing an example of how application code might be written

• SKI Assets (Server Side):
  o Java Classes—providing assistance in formulating RESTful Responses

Figure 24 SDN Controller main UI

SKI Framework - Navigation Tree

The SKI framework implements a navigation model consisting of a list of top-level categories in which each category consists of a list of navigation items. Each navigation item consists of a list of views in which one of the views is considered the default View. The default View is selected when the navigation item is selected. The other views associated with the navigation item can be navigated to using the selector buttons located on the view toolbar. Figure 25 shows the SKI UI view diagram.
SKI Framework - Hash Navigation

The SKI Framework encodes context and navigation information in the URL hash. For example, consider the URL:

http://appserver.rose.hp.com/webapp/ui/app/#hash

The #hash portion of the URL is encoded as #vid,ctx,sub, where:

- vid—is the view ID, used to determine which view to display
- ctx—is the context, used to determine what data to retrieve from the server
- sub—is the sub-context, used to specific any additional context information with respect to the view (that is, select a specific row in a table)

The following diagrams show the sequence of events on how SKI selects a view and loads the data if a URL is pasted into the browser. The #hash is decoded into #vid,ctx,sub, as shown in Figure 26. The vid (view ID) is used to determine the view, navigation item and category to be selected.
Next, the $ctx$ (context), shown in Figure 27, can be used to help determine what data to retrieve from the Server RESTlet.

When the Asynchronous HTTP request returns, the data (likely in JSON form), as shown in Figure 28, can be used to populate the view’s DOM (grids, widgets, etc.).
Finally, the sub (sub-context) can be used to specify additional context information to the view. In this case, the second item is selected, as shown in Figure 29.

**Figure 29** SKI UI view sub-context hash diagram
SKI Framework - View Life-Cycle

All views are event driven and can react to the following life-cycle events:

- **Create**—called a single time when the view needs to be created (that is, navigation item is clicked for the first time). At this time, a view will return its created DOM structure (that is, an empty table).
- **Preload**—called only once, after the view is in the DOM. At this time, a view can perform any initialization that can only be done after the DOM structure has been realized.
- **Reset**—may be called multiple times, allows the view to clear any stale data
- **Load**—may be called multiple times, allows the view to load its data. This is where a view can make any Ajax calls needed to obtain server-side data.
- **Resize**—may be called multiple times, allows the view to handle resize events caused by the browser or main layout
- **Error**—may be used to define an application specific error handler for the view
- **Unload**—called to allow a view to perform any cleanup as it is about to be replaced by another view

SKI Framework - Live Reference Application

The SKI reference application `hp-util-ski-ui-X.XX.X.war` is distributed with the SDK in the `lib/util/` directory. You need to install the Apache Tomcat web server to run the reference application. Simply copy this war file into your Tomcat `webapps` directory as the file `ski-ui.war`. You can launch the reference application in your browser with URL: `localhost:8080/ski-ui/ref/index.html`.

**Figure 30** shows the SKI UI reference application.
From these pages, you have access to the most up to date documentation and reference code. The reference application includes examples on how to:

- Add categories, navigation items and views.
- Create a jQuery UI layout in your view.
- Create various widgets (buttons, radios, and so on) in your view.

**UI Extension**

The SDN UI Extension framework allows third-party application to inject UI content seamlessly into the main SDN UI. The following list is the important files a developer needs to be aware of to make use of the UI Extensions framework. For more information, see Distributed Coordination Primitives see 5 Sample Application.
Introduction

In a network managed by a controller, the controller itself stands out to be a single point of failure. Controller failures can disrupt the entire network functionality. HP VAN SDN Controller Distributed Coordination infrastructure provides various mechanisms that controller applications can make use of in achieving active-active, active-standby Distributed Coordination paradigms and internode communication. The Distributed Coordination infrastructure provides 2 services for the applications to develop Distributed Coordination aware controller modules.

- Controller Teaming
- Distributed Coordination Service

Following figure describes the communication between the controller applications and the HP VAN SDN Controller Distributed Coordination sub-systems. “App1 – 1” indicates instance of application 1 on controller instance 1. Distributed services, ensures the data synchronization across the controller cluster nodes.
Controller Teaming

Teaming Configuration Service

The Teaming Configuration Service provides REST interfaces (/team) that can be used to set up a team of controllers. Without team configuration, controller nodes will bootstrap in standalone mode. As the teaming is configured, identified nodes form a cluster and the controller Applications can communicate across the cluster using Coordination Service interfaces.

The following curl command is used to get the current team configuration. 192.168.66.1 is the IP address of one of the teamed controllers.

```
curl --noproxy 192.168.66.1 --header "X-Auth-Token: 19a4b8a048ef4965882eb8c570292bcd" --request GET --url https://192.168.66.1:8443/sdn/v2.0/team -ksS
```

For team creation help and other configuration commands please refer to HP VAN SDN Controller Administrator Guide [29].

Distributed Coordination Service

Distributed Coordination Service provides the building blocks to achieve high availability in the HP VAN SDN Controller environment. This service can be retrieved from the Teaming Service. An example java application that makes use of different functionalities of the Coordination Service is described in the subsequent sections.

Distributed Coordination Service includes:

- Publish Subscribe Service
- Distributed Maps
- Distributed Locks

Serialization

It is required to register a Serializer for each distributable object because of the multiple class loaders approach followed by OSGi. No serializer is required for the following types: Byte, Boolean, Character, Short, Integer, long, Float, Double, byte[], char[], short[], int[], long[], float[], double[], String.

If a distributable object implements Serializable, Distributable must be found before Serializable in the class hierarchy going from the distributable object to its super classes. Unfortunately the order matters: The class hierarchy is analyzed when registering the serializer. If Serializable is found before Distributable an exception is thrown with a message describing this restriction.

Example of distributable object declarations:
import com.hp.api.Distributable

class ValidDistributableType implements Distributable {
}

class ValidDistributableType implements Distributable, Serializable {
}

class ValidDistributableType extends SerializableType implements Distributable {
}

class InvalidDistributableType implements Serializable, Distributable {
}

Example of serializer registration:

```java
@Component
public class Consumer {
    @Reference(cardinality = ReferenceCardinality.MANDATORY_UNARY, policy = ReferencePolicy.DYNAMIC)
    private volatile CoordinationService coordinationService;

    @Activate
    public void activate() {
        coordinationService.registerSerializer(new MyDistributableObjectSerializer(), MyDistributableObject.class);
    }

    @Deactivate
    public void deactivate() {
        coordinationService.unregisterSerializer(MyDistributableObject.class);
    }

    private static class MyDistributableObjectSerializer implements Serializer<MyDistributableObject> {
        @Override
        public byte[] serialize(MyDistributableObject subject) {
            ...
        }

        @Override
        public MyDistributableObject deserialize(byte[] serialization) throws IllegalArgumentException {
            ...
        }
    }
}
```

**Publish Subscribe Service**

In a distributed environment, applications tend to communicate with each other. Applications might be co-located on the same controller node or they may exist on different nodes of the same controller cluster. The Publish Subscribe Service provides a way to accomplish this kind of distributed communication mechanism. Note that communication can occur between the nodes of a controller cluster and not across controller clusters. The Publish Subscribe Service provides a mechanism where several applications on different controller nodes can register for various types of bus messages, send and receive messages without worrying about delivery failures or out of order delivery. When an application pushes a message, all the subscribers to that message type for active members of the team are notified irrespective of their location in the controller cluster.
Publish Subscribe service is provided by the Distributed Coordination Service which is in turn provided by the Teaming service. Please refer to the Javadoc for a detailed explanation of methods provided by publish-subscribe service.

Publish Subscribe service also provides mechanisms to enable global ordering for specific message types. Global ordering is disabled by default. With global ordering enabled, all receivers will receive all messages from all sources with the same order. If global order is disabled two different receivers could receive messages from different sources in different orders. It is important to note - since global ordering degrades performance - that messages from the same source will still be ordered even with global ordering disabled.

Example:

Let A and B be message publishers (Sources).

Let R and W be message subscribers (Receivers).

Assume A sends messages a1 a2 a3 in that order.

Assume B sends messages b1 b2 b3 in that order.

With or without global ordering the following holds:

- a1 arrives before a2
- a2 arrives before a3
- b1 arrives before b2
- b2 arrives before b3

With global ordering

- Let a1 b1 a2 a3 b2 b3 be the sequence of messages received by R
- Then W receives messages in the same order

Without global ordering

- Let a1 b1 a2 a3 b2 b3 be the sequence of messages received by R
- Then W may (or may not) receives messages in the same order.

The global ordered sequence does not necessarily represent the sequence in which the events were actually generated, but the sequence in which they were received by a node designated as a reference automatically by the Distributed Coordination service. This reference node propagates the events in the order received; this is how global ordering is commonly implemented. Thus, global ordering is from the receiving point of view and not from the sending point of view (It is not possible to determine the actual order events were generated - common problem in distributed systems: It is not possible to get a global state of the system).

The example below presents a common implementation of publish subscribe service.

Publish-Subscribe Example:

PubSubExample.java

```
import com.hp.sdn.teaming.TeamingService;
```
import com.hp.util.dcord.CoordinationService;
import com.hp.sdn.demo.example.SampleMessage;

@Component
public class PubSubExample {
    private CoordinationService coordinationService;
    private PublishSubscribeService pubSubService;

    @Reference(cardinality = ReferenceCardinality.MANDATORY_UNARY, policy =
        ReferencePolicy.DYNAMIC)
    protected volatile TeamingService teamingSvc;

    @Activate
    protected void activate() {
        coordinationService = teamingSvc.getCoordinationService();
        pubSubService = coordinationService.getPublishSubscriberService();
    }

    public void subscribe() {
        SampleMessageListener<SampleMessage> listener = new
        SampleMessageListener<SampleMessage>();
        pubSubService.subscribe(listener, SampleMessage.class);
    }

    public void publish(SampleMessage message) {
        pubSubService.publish(message);
    }
}

Message Listener Example:

SampleMessageListener.java
import com.hp.util.dcord.MessageEvent;
import com.hp.util.dcord.Subscriber;
import com.hp.sdn.demo.example.SampleMessage;

public class SampleMessageListener<M extends SampleMessage> implements
    Subscriber<M> {
    @Override
    public void onMessage(MessageEvent<M> messageEvent) {
        // Any action to be taken on receipt of a message notification.
        // In this example, there is a simple print
        System.out.println("Message notification received");
    }
}
Distributed Map

A Distributed Map is a class of a decentralized distributed system that provides a lookup service similar to a hash table; (key, value) pairs are stored in a Distributed Map, and any participating node can efficiently retrieve the value associated with a given key. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. This allows a Distributed Map to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures.

The distributed map is an extension to the Java Map interface and due to this, the applications can perform any operation that can be performed on a regular Java map. The data structure internally distributes data across nodes in the cluster. The data is almost evenly distributed among the members and backups can be configured so the data is also replicated. Backups can be configured as synchronous or asynchronous; for synchronous backups when a map.put(key, value) returns, it is guaranteed that the entry is replicated to one other node. Each distribute map is distinguished by the namespace and it is set upon creation of the distributed map.

The Distributed Coordination Service provides a mechanism where applications running on multiple controllers to register for notifications for specific distributed maps. Notifications of a distributed map are received when entries in the distributed map are added, updated or removed. Notifications are received per entry.

Distributed Map Example:

```java
package com.hp.dcord_test.impl;

import java.util.Map.Entry;
import org.apache.felix.scr.annotations.Activate;
import org.apache.felix.scr.annotations.Component;
import org.apache.felix.scr.annotations.Reference;
import org.apache.felix.scr.annotations.ReferenceCardinality;
import org.apache.felix.scr.annotations.ReferencePolicy;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.hp.sdn.teaming.TeamingService;
import com.hp.util.dcord.CoordinationService;
import com.hp.util.dcord.DistributedMap;
import com.hp.util.dcord.Namespace;

@Component
public class SampleDistributedMap {
    private CoordinationService coordinationService;
    private SimpleEntryListener listener;

    @Reference(cardinality = ReferenceCardinality.MANDATORY_UNARY,
        policy = ReferencePolicy.DYNAMIC)
```
protected volatile TeamingService teamingSvc;

@Activate
protected void activate() {
    coordinationService = teamingSvc.getCoordinationService();
}

public void createDistributedMap(String namespace) {
    Namespace mapNamespace = Namespace.valueOf(namespace);

    DistributedMap<String, String> distMap =
        coordinationService.getMap(mapNamespace);

    if (distMap == null) {
        throw new RuntimeException("Can't get a Distributed Map instance.");
    }
}

public void deleteDistributedMap(String namespace) {
    Namespace mapNamespace = Namespace.valueOf(namespace);

    DistributedMap<String, String> distMap =
        coordinationService.getMap(mapNamespace);

    if (distMap == null) {
        throw new NullPointerException();
    }

    distMap.clear();
}

public void readDistributedMap(String namespace) {
    Namespace mapNamespace = Namespace.valueOf(namespace);

    DistributedMap<String, String> distMap =
        coordinationService.getMap(mapNamespace);

    if (distMap == null) {
        throw new RuntimeException("Can't get a Distributed Map instance.");
    }

    for (Entry<String, String> entry : distMap.entrySet()) {
        String stringKey = "key " + entry.getKey().toString();
        System.out.println(stringKey);
        String stringValue = "value " + entry.getValue().toString();
        System.out.println(stringValue);
    }
}
public void writeDistributedMap(String namespace, String key, String value) {
    ObjectMapper mapper = new ObjectMapper();
    Namespace mapNamespace = Namespace.valueOf(namespace);

    DistributedMap<String, String> distMap = coordinationService.getMap(mapNamespace);
    if(distMap == null){
        throw new RuntimeException("Can't get a Distributed Map instance.");
    }

    distMap.put(key, value);
}

public void subscribeListener(String namespace) {
    Namespace mapNamespace = Namespace.valueOf(namespace);

    DistributedMap<String, String> distMap = coordinationService.getMap(mapNamespace);
    if(distMap == null){
        throw new RuntimeException("Can't get a Distributed Map instance.");
    }

    listener = new SimpleEntryListener();
    if (listener == null) {
        throw new RuntimeException("Can't get a SimpleEntryListener instance.");
    }

    distMap.register(listener);
}

public void unSubscribeListener(String namespace) {
    Namespace mapNamespace = Namespace.valueOf(namespace);

    DistributedMap<String, String> distMap = coordinationService.getMap(mapNamespace);
    if(distMap == null){
        throw new RuntimeException("Can't get a Distributed Map instance.");
    }
}
if (listener != null) {
    distMap.unregister(listener);
}

SimpleEntryListener.java
package com.hp.dcord_test.impl;

import com.hp.util.dcord.EntryEvent;
import com.hp.util.dcord.EntryListener;

public class SimpleEntryListener implements EntryListener<String, String> {

    @Override
    public void added(EntryEvent<String, String> entry) {
        // Any action to be taken on receipt of a message notification.
        // In this example, there is a simple print
        String string = "Added notification received";
        System.out.println(string);
    }

    @Override
    public void updated(EntryEvent<String, String> entry) {
        // Any action to be taken on receipt of a message notification.
        // In this example, there is a simple print
        String string = "Updated notification received";
        System.out.println(string);
    }

    @Override
    public void removed(EntryEvent<String, String> entry) {
        // Any action to be taken on receipt of a message notification.
        // In this example, there is a simple print
        String string = "Removed notification received";
        System.out.println(string);
    }
}

Performance Considerations
Keep in mind the following when using the distributed coordination services:
1. Java objects can be written directly to distributed coordination services.
   - There is no need to serialize the data before it is written to these structures.
   - The coordination service will serialize/deserialize the data as it is distributed in the team
   using the serializer you have registered.

2. Minimize other in-memory local caches for distributed map data.
3. Minimize tying map entry listeners to persistence.
- Consider how important it is for your data to be persisted before automatically tying a distributed map entry listener for the purpose of writing to the database.

### Distributed Lock

Protecting the access to shared resources becomes increasingly important in a distributed environment. A lock is a synchronization primitive that ensures only a single thread is able to access a critical section. Distributed Locks offered by the Coordination Service provides an implementation of locks for distributed environments where threads can run either in the same JVM or in different JVMs.

Applications needs to define a namespace that is used as the lock identity to make sure application instances running on different JVMs acquire the right lock. Applications on different controller nodes should agree upon the namespace and acquire the necessary lock on it before accessing the shared resource.

A distributed lock extends the functionality of java.util.concurrent.locks.Lock and thus it can be used as a regular Java lock with the following differences:

- Locks are automatically released when a member (node) has acquired a lock and this member goes down. This prevents threads that are waiting for a lock from waiting indefinitely. This is needed for failover to work in a distributed system. The downside however is that if a member goes down that acquired the lock and started making changes, other members could start to see partial changes.

**Distributed Lock Example:**

```java
Namespace namespace = Namespace.forReplicatedProcess(getClass());
Lock lock = coordinationService.getLock(namespace);
lock.lock();
try {
    // access the resources protected by this lock
} finally {
    lock.unlock();
}
```

### Data Versioning with Google Protocol Buffers (GPB)

For the long term maintainability, interoperability, and extensibility of application data it is recommended that applications version the data they write using the different coordination services. Google Protocol Buffers (GPB) is the recommended versioning mechanism for these services that is supported by the SDK. The section below introduces GPBs and their use for message versioning with application’s model objects. It is recommended the reader reference the official GPB documentation to understand the complete syntax and all the features available for the programming language of choice for your application. [50]
GPB is a strongly-typed Interface Definition Language (IDL) with many primitive data types. It also allows for composite types and namespaces through packages. Users define the type of data they wish to send/store by defining a protocol file (.proto) that defines the field names, types, default values, requirements, and other metadata that specifies the content of a given record. [50, 51]

Versioning is controlled in the .proto IDL file through a combination of field numbers and tags (REQUIRED/OPTIONAL/REPEATED). These tags designate which of the named fields must be present in a message to be considered valid. There are well-known rules of how to design a .proto file definition to allow for compatible versions of the data to be sent and received without errors (see Versioning Rules section that follows).

From the protocol file a provided Java GPB compiler (protoc) then generates the data access classes for the user’s language of choice. In the generated GPB class, field access and builder methods are provided for the application to interact with the data. The compiler also enforces the general version rules of messages to help flag not only syntax and semantic error, but also errors related to incompatibility between versions of a message.

The application will ultimately use the Model Object it defines and maps to the GPB class that will be distributed. The conversion from Model Object to GPB object takes place in the custom serializer the programmer will have to write and register with the Coordination Service to bridge the object usage in the application and its distribution over the Coordination Services (See Application GPB Usage section that follows for more details).

Below is an example of a GPB .proto file that defines a Person by their contact information and an AddressBook by a list of Persons. This example demonstrates the features and syntax of a GPB message. String and int32 are just two of the 15 definable data types (including enumerated types) which are similar to existing Java primitive types. Each field requires a tag, type, name, and number to be valid. Default values are optional. Message structures can be composed of other messages. In this example we see that a name, id and number are the minimum fields required to make up a valid Person record. If this were version 1 of the message then, for example, version 2 could include an “optional string website = 5;” field to expand the record further without breaking compatibility with version 1 of the Person record. The Addressbook message defines a composition of this Person message to hold a list of people using the repeated tag. [51]

```proto
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }
  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }
}
```
The protocol file above would be run through GPB’s Java compiler (See “.proto Compilation Process” Below) to generate the data access classes to represent these messages. Message builders would allow new instances of the message to be created for distribution by the Coordination Services. Normal set/get accessor methods will also be provided for each field. Below are examples of creating a new instance of the message in Java. Reading the record out will return this GPB generated object for the application to interact with as usual.

```
public class AddPerson {

    // This function creates a simple instance of a GPB Person object
    // that can then be written to one of the Coordination Services.
    public static Person createTestPerson() {
        // Initial GPB instance builder.
        Person.Builder person = Person.newBuilder();

        // Set REQUIRED Person fields.
        person.setName("John Doe");
        person.setId("1234");

        // Set OPTIONAL Person fields.
        person.setEmail("john.doe@gmail.com");

        // Set REQUIRED Phone fields.
        Person.PhoneNumber.Builder phoneNumber = 
            Person.PhoneNumber.newBuilder().setNumber("555-555-5555");
        phoneNumber.setType(Person.PhoneType.MOBILE);
        person.addPhone(phoneNumber);

        return person.build();
    }

    // Versioning Rules
```
A message version is a function of the field numbering and tags provided by GPB and how those are changed between different iterations of the data structure. The following are general rules about how .proto fields should be updated to insure compatible GPB versioned data:

- Do not change the numeric tags for any existing (previous version) fields.
- New fields should be tagged OPTIONAL/REPEATED (never REQUIRED). New fields should also be assigned a new, unique field ID.
- Removal of OPTIONAL/REPEATED tagged fields are allowed and will not affect compatibility.
- Changing a default value for a field is allowed. (Default values are sent only if the field is not provided.)
- There are specific rules for changing the field types. Some type conversions are compatible while others are not (see GPB documentation for specific details).

Note: It is generally advised that the minimal number of fields be marked with a REQUIRED tag as these fields become fixed in the schema and will always have to be present in future versions of the message.

.proto Compilation Process

The following is a description of the process by which .proto files should be defined for an application, compiled with the Java GPB compiler, and how the derived data classes should be imported and used in application code. Application developers that wish to make use of GPB in their designs will need to download and install Google Protocol Buffers (GPB) on their local development machine. Those steps are as follows for GPB 2.5.0v:

Compiling and installing the protoc binary

The protoc binary is the tool used to compile your text-based .proto file into a source file based on the language of your choice (Java in this example). You will need to follow these steps if you plan on being able to compile GPB-related code.

1. Download the “full source” of Google’s Protocol Buffers. For this example we are using 2.5.0v in the instructions below.
2. Extract it somewhere locally.
3. Run the following line:
   ```bash
cd protobuf-2.5.0
./configure && make && make check && sudo make install
```
4. Add the following to your shell profile and also run this command:
   ```bash
   export LD_LIBRARY_PATH=/usr/local/lib
   export LD_LIBRARY_PATH=/usr/local/lib
   ``
5. Try to run it standalone to verify protoc is in your path and the LD_LIBRARY_PATH is set correctly. Running “protoc” on the command line should return “Missing input file.” If everything is setup correctly.

Compiling .proto Files
We recommend under the project you wish to define and use GBP you place .proto files under the /src/main/proto directory. You can then make use of the GBP “option java_package” syntax to control the subdirectory/package structure that will be created for the generated Java code from the .proto file.

The projects pom.xml file requires the following GPB related fields:

```xml
<dependencies>
  <dependency>
    <groupId>com.google.protobuf</groupId>
    <artifactId>protobuf-java</artifactId>
    <version>2.5.0</version>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.3.2</version>
      <configuration>
        <source>1.7</source>
        <target>1.7</target>
      </configuration>
    </plugin>
    <plugin>
      <groupId>com.google.protobuf.tools</groupId>
      <artifactId>maven-protoc-plugin</artifactId>
      <version>0.3.2</version>
      <executions>
        <execution>
          <goals>
            <goal>compile</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

After running “mvn clean install” on the pom.xml file GPB’s protoc will be used to:

- Generate the necessary Java files under:
  ./target/generated-sources/protobuf/java/<optional java_package directory>
- Compile the generated Java file into class files
· Package up the class files into a jar in the target directory
· Install the compiled jar into your local Maven cache (~/.m2/repository)

Have the .proto file and generated .java file displayed properly in your IDE from your project’s root directory, i.e. where the project’s pom.xml file is, execute the following:
· mvn eclipse:clean
· mvn eclipse:eclipse
· Refresh the project in your IDE (Optional: clean the project as well).

As the resulting Java file is protoc generated code it is not recommended that it be checked in to your local source code management repo but instead regenerated when the application is built. The GPB Java Tutorial link on the official GPB website gives a more in depth walk through of the resulting Java class.

Application GPB Usage

Generated GPB message classes are meant to serve as the versioned definition of data distributed by the Coordination Service. They are not meant to be used directly by the application to read/write to the various Coordination Services. It is recommended that a Model Object be defined for this role. This scheme provides two notable benefits:

1) It allows the application to continue to evolve without concern for the data versioning at the Coordination Service level.

2) It allows the Model Object to define fields for data it may want to store and use locally for a version of the data but not have that data shared during distribution.

The recommended procedure for versioning Coordination Service data is shown below and the sections that follow explain each of these steps with examples and best practices.

1) Define a POJO Model Object for the data that the application will want to operate on and distribute via a Coordination Service.

2) Define a matching GPB .proto Message to specify which field(s) of the Model Object are required/optional for a given version of message distributed by the Coordination Services.

3) Implement and register a Custom Serializer with the Coordination Service that will convert the Model Object the application uses to the GPB message class that will be distributed.

Model Object

The application developer will define POJOs for his/her application. They will contain data and methods necessary to the applications processing and may contain data that the application wishes to distribute to other members of the controller team. Not all fields may need to be (or want to be) distributed. The only requirement for the Model Object’s implementation is that the class being written to the different Coordination Services
implement com.hp.api.Distributable (a marker interface) to make it compatible with the Coordination Service.

In terms of sharing these objects via the Coordination Service, the application developer should consider which field(s) are required to constitute a version of the Model Object versus which fields are optional. Commonly those fields that are defined in the objects constructor arguments can be considered required fields for a version of the object. Later versions may add additional optional fields to the object that are not set by a constructor. New required fields may be added for new versions of the Model Object with their presence as an argument in a new constructor. Note that adding new required fields will require that field for future versions. Past versions of the application that receive a new required field will just ignore it. Overall, thinking in terms of what fields are optional or required will help with the next step in the definition of the GPB .proto message.

The following is an example of a Person Java class an application may want to define and distribute via a PubSub Message Bus. The name and id fields are the only required as indicated with the constructor arguments. The application may use other ways to indicate what required fields are.

```java
class Person implements Distributable {
    private String name;
    private int id;

    private String email;
    private Date lastUpdated;

    Person(String name, Id id) {
        this.name = name;
        this.id = id;
    }
    // Accessor and other methods.
}
```

**GPB .proto Message**

The GPB .proto message serves as the definition of a versioned message to be distributed by the Coordination Service. The application developer should write the .proto messages with the Model Object in mind when considering the data type of fields, whether they are optional or required. etc. The developer should consider all the GPB versioning rules and best practices mentioned in the previous section. The programmer implements a message per Model Object that will be distributed following the GPB rules and conventions previously discussed.

Below is an example .proto message for the Person class. The field data types and REQUIRED/OPTIONAL tags match the Model Object. Since email was not a field to be set in the constructor it is marked optional while name and id are marked as required. Notice that lastUpdated field of the Model Object is not included in the .proto message definition. This is considered a transient field, in the serialization sense, for the Model Object and it is not meant to be distributed in any version of the message. With this example the reader can
see not all fields in the Person Model Object must be defined and distributed with the .proto
message.

```protobuf
go
option java_outer_classname = "PersonProto; // Wrapper class name.

message Person {  
  required string name = 1;  
  required int32 id = 2;  
  optional string email = 3;  
}
```

The application developer will generate the matching wrapper and builder classes for the
.proto message to have a Java class that defines the message using protoc in the .proto
Compilation Process section above.

**Custom Serializer**

Finally, a custom serializer needs to be defined to translate between instances of the
Model Object being used in the Coordination Services and instances of the GPB message
that will ultimately be transported by that service. For example, we may wish to write the
Person Model Object on the PubSub Message Bus and have it received by another instance
of the application which has subscribed to Person messages through its local Coordination
Service.

In the custom serializer the developer will map the fields between these two objects on
transmit (serialization) and receive (deserialization). With data types and naming
conventions it should be clear what this 1:1 mapping is in the serializer. The Serializer must
implement the Serializer<Model Object> interface as shown in the example below. It is
recommended this serializer be kept in the <application>-bl project (if using the provided
application project generation script of the SDK). PersonProto is the java_outer_classname
we define in the GPB message above and will be the outer class from which inner GPB
message classes, and their builders, are defined.

```java
import <your package>.PersonProto;

public class PersonSerializer implements Serializer<Person> {  
  @Override  
  public byte[] serialize(Person subject) {  
    PersonProto.Person.Builder message = PersonProto.Person.newBuilder();  
    message.setName(subject.getName());  
    message.setId(subject.getId());  
    return message.build().toByteArray();  
  }  

  @Override  
  public Person deserialize(byte[] serialization) {  
```
PersonProto.Person message = null;
try {
    message = PersonProto.Person.parseFrom(serialization);
} catch (InvalidProtocolBufferException e) {
    // Handle the error
}

Person newPerson = new Person();
if (message != null) {
    newPerson.setName(message.getName());
    newPerson.setId(message.getId());
    return newPerson;
}
return null;
}

In the serialize() method the builder pattern of the generated GPB message class is used to create a GPB version of the Person Model Object. After the proper fields are set the message is built and converted to a byte array for transport. In the deserialize() method on the receiver the byte array is converted back to the expected GPB message object. An instance of the Model object is then created and returned to be placed into the Coordination Service for which the serializer is registered.

The application must register this custom serializer with the Coordination Service it wishes to use this Model Object and GPB message combination. Below is an example of that registration process in an OSGI Component of an example application.

```java
@Reference(cardinality = ReferenceCardinality.MANDATORY_UNARY,
            policy = ReferencePolicy.DYNAMIC)
protected volatile CoordinationService coordinationSvc;

@Activate
public void activate() {
    // Register Message Serializers
    if (coordinationSvc != null) {
        coordinationSvc.registerSerializer(new PersonSerializer(),Person.class);
    }
}
```

**System Status**

The system status (Which can be retrieved using SystemInformationService) depends on two properties of the controller: Reachability and Health. The following table depicts the status:

<table>
<thead>
<tr>
<th>Property</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reachability</td>
<td>Healthy</td>
</tr>
<tr>
<td>Health</td>
<td>Healthy</td>
</tr>
</tbody>
</table>

83
### Table 3 System Status

<table>
<thead>
<tr>
<th>System Status</th>
<th>Coordination Services</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>Available</td>
<td>The controller is healthy and part of a cluster with a quorum.</td>
</tr>
<tr>
<td>Suspended</td>
<td>Unavailable</td>
<td>The controller is unhealthy or part of a cluster with no quorum.</td>
</tr>
<tr>
<td>Unreachable</td>
<td>Depends whether active or suspended</td>
<td>The controller is unreachable because of failures or network partition.</td>
</tr>
</tbody>
</table>

Considerations:
- A system never sees itself as unreachable.
- The strategy followed on the event of a network partition is to suspend controllers that are part of a cluster with no quorum.

The following figure illustrates two examples of how each controller sees the status of the other controllers that are part of the team. Examples show a 5-node cluster for simplicity; this does not mean this release supports teams of such size. The behavior shown in the examples can easily be applied to any cluster size.
Persistence

Distributed Persistence Overview

The SDN Controller provides a distributed persistence for applications in form of a Cassandra [10] database node running on each controller instance. A team of controllers serves as a Cassandra cluster. Cassandra provides the following benefit as a distributed database:

- A distributed, peer-to-peer datastore with no single point of failure.
- Automatic replication of data for improved reliability and availability.
- An eventually-consistent view of the database from any node in the cluster.
- Incremental, scale-out growth model.
- Flexible schemas (column oriented keyspaces).
- Hadoop integration for large-scale data processing.
- SQL-like query support via Cassandra Query Language (CQL).

Distributed Persistence Use Case

The distributed persistence architecture is targeted at applications that have distributed active-active requirements. Specifically, applications should use the distributed persistence framework if they have one or more of following requirements:
• Consumer applications have high scalability requirements i.e. there are generally multiple instances of the app running on different controller nodes that need access to a common distributed database store.
• The distributed database should be available independent of whether individual nodes are present or not e.g. if there are controller node crashes.
• The applications have high throughput requirements: large number of I/O operations. Further, they have requirements wherein as the number of controller nodes increases, performance needs to scale linearly.

For addressing applications with such requirements, a distributed persistence layer that uses Cassandra is exported as the underlying distributed database. The HP VAN SDN Controller provides a Data Access Object (DAO) layer on top of Cassandra for performing distributed persistence operations.

**Persistence Data Model**

**Introduction to DAO Pattern**

A data access object (DAO) is an object that provides an abstract interface to some type of database or persistence mechanism, providing some specific operations without exposing details of the database. It provides a mapping from application calls to the persistence layer. This isolation separates the concerns of what data accesses the application needs, in terms of domain-specific objects and data types (the public interface of the DAO), and how these needs can be satisfied with a specific DBMS, database schema, and so on. Figure 33 and Figure 34 show Data Access Object pattern [30].

**Figure 33 Data Access Object Pattern**
Distributed Data Model Overview

Cassandra is a “column oriented” distributed database system and provides a structured key-value store. It is a NOSQL database and this means it is completely non-relational in nature. A reference table which can be useful for migration of a MySQL (RDBMS) to a NOSQL DB (Cassandra) is as illustrated in Figure 35.

Figure 35 Mental Model Comparison between Relational Models and Cassandra

<table>
<thead>
<tr>
<th>Relational Model</th>
<th>Cassandra Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>Keyspace</td>
</tr>
<tr>
<td>Table</td>
<td>Column Family (CF)</td>
</tr>
<tr>
<td>Primary key</td>
<td>Row key</td>
</tr>
<tr>
<td>Column name</td>
<td>Column name/key</td>
</tr>
<tr>
<td>Column value</td>
<td>Column value</td>
</tr>
</tbody>
</table>

Although this table provides a mapping of the terms, a more accurate analogy is a nested sorted map. Cassandra stores data in the format as follows:

\[
\text{Map<RowKey, SortedMap<ColumnKey, ColumnValue>>}
\]

So, there is a sorted map of RowKeys to an internal Sorted map of Columns sorted by the ColumnKey. The following figure illustrates a Cassandra row.
This is a simple row with columns. There are other variants like Composite Columns and Super Columns which allow more levels of nesting. These can be visited if there is a need for these in the design.

One important characteristic of Cassandra is that it is schema-optional. This means the columns need not be defined upfront. They can be added dynamically as and when required and further all rows need not have the same number and type of columns.

Some important points to be noted during migration of data from RDBMS to NOSQL are as follows:

- Model data with nested sorted maps in mind as mentioned above. This provides an efficient and faster response time for queries.
- Model Column families around queries.
- De-normalize data as needed. Too much of de-normalization can have side effects. A right balance needs to be struck.

**Modeling Data around Queries**

Unlike with relational systems, where entities and relationships are modeled and then indexes are added to support whatever queries become necessary, with Cassandra queries that need to be supported efficiently are thought of ahead of time.

Cassandra does not support joins at the query time because of its high scale distributed nature. This mandates duplication and de-normalization of data. Every column family in a Cassandra keyspace is self-contained with all data necessary to satisfy a given query. Thus, moving towards a “Column Family per query” model.

In the HP VAN SDN Controller, define a column family for every entity. For each query on that entity, define a secondary column family. These secondary column families serve exactly one query.

**Reference Application using Distributed Persistence**

Any application that needs to use the distributed persistence in the HP VAN SDN Controller needs to include/define the following components:

- A Business Logic component as an OSGi service.
- A reference to Distributed DataStoreService and Distributed QueryService
- A DTO (transport object) per entity.
- DAO—Data access object to interact with the persistence layer.
- A sample of each of these will be presented in this section. For demonstration purposes a Demo application that persists Alerts in the Distributed Database (Cassandra) has been created.

**Business Logic Reference**

When the Cassandra demo application is installed, the OSGi service for business logic gets activated. This service provides a north bound interface. Any external entity/app can use this service via the API provided by this service. In this case, we have Alert service using Cassandra. This service provides an API for all north bound operations such as posting an Alert into the database, deleting the alerts and updating the alert state. There is another interface that provides for the READ operations and is mostly used by the GUI interface. This second north bound service is called CassandraAlertUIService.

The implementation of these services needs to interact with the underlying persistence layer. This is done by using an OSGi @Reference as shown below.

**CassandraAlertManager.java:**

```java
@Component
@Service
public class CassandraAlertManager implements CassandraAlertUIService, CassandraAlertService {

    @Reference(policy = ReferencePolicy.DYNAMIC,
               cardinality = ReferenceCardinality.MANDATORY_UNARY)
    private volatile DataStoreService<DataStoreContext> dataStoreService;

    @Reference(policy = ReferencePolicy.DYNAMIC,
               cardinality = ReferenceCardinality.MANDATORY_UNARY)
    private volatile DistQueryService<DataStoreContext> queryService;

    ...  
}
```

The above snippet shows the usage of @Reference. OSGi framework caches the dataStoreService and queryService objects in the CassandraAlertManager. Whenever, the client or application issues a query to the database, these objects will be used to get access to the persistence layer.

**DTO (Transport Object)**

Data that needs to be persisted can be divided into logical groups and these logical groups are tables of the database. Every table has fixed columns and every row has a fixed type of Row Key or Primary Key.

DTO is a java representation of a row of a table in the database. Any application that needs to write a row needs to fill data into a DTO and hand it over to the persistence layer. The persistence layer understands a DTO and converts it into a format that is required for the underlying database. The reverse holds too. When reading something from the database, the data will be converted into a DTO (for a single row read) or a list of DTO (multi row read) or a page of DTO (paged read) and given back to the requestor.

Here is an example DTO used in the demo app:

**CassandraAlert.java:**
package com.hp.demo.cassandra.model.alert;

import com.hp.api.Id;
import com.hp.demo.cassandra.model.AbstractTransportable;
...
public class CassandraAlert extends AbstractTransportable<CassandraAlert, String> { 
...
private Severity severity;
private Date timestamp;
private String description;
private boolean state;
private String origin;
private String topicId;

public CassandraAlert(String sysId, boolean state, String topicId, String origin, Date timestamp, Severity severity, String description) {
    super(sysId);
    init(topicId, origin, timestamp, severity, state, description);
}

public CassandraAlert(String uid, String sysId, boolean state, String topicId, String origin, Date timestamp, Severity severity, String description) {
    super(uid, sysId);
    init(topicId, origin, timestamp, severity, state, description);
}

public CassandraAlert(String uid) {
    super(uid, null);
}

@Override
public Id<CassandraAlert, String> getId() {
    return Id.<CassandraAlert, String>valueOf(this.uid());
}

// Implement getters for immutable fields.
// Implement setters and getters for mutable fields.

// Good practice to override the following methods on transport objects:
// equals(Object), hashCode() and toString()
...}
The function of a DTO is to list out all the columns and provide setters/getters for each of the attributes. The application fills out all the values as necessary and passes the object down to the persistence layer using various queries.

**Distributed Database Queries**

The distributed persistence layer of the HP VAN SDN Controller exposes the following queries to the application:

- AddQuery
- CountQuery
- DeleteQuery
- DeleteQueryWithFilter
- FindQuery
- GetQuery
- PagedFindQuery
- UpdateQuery

These are generic queries and need to be qualified appropriately by the application. The following shows a Distributed Query Service interface that provides application specific queries.

Here is the interface code from the demo application.

DistQueryService.java:

```
package com.hp.demo.cassandra.dao;

import com.hp.demo.cassandra.model.alert.CassandraAlert;
import com.hp.demo.cassandra.model.alert.CassandraAlertFilter;
import com.hp.demo.cassandra.model.alert.CassandraAlertSortAttribute;
import com.hp.util.MarkPage;
import com.hp.util.MarkPageRequest;
import com.hp.util.SortSpecification;
import com.hp.util.persistence.ReadQuery;
import com.hp.util.persistence.WriteQuery;
...
public interface DistQueryService<C> {

    ReadQuery<List<CassandraAlert>, C> getFindAlertsQuery(CassandraAlertFilter filter,
                SortSpecification<CassandraAlertSortAttribute> sortSpecification);

    ReadQuery<MarkPage<CassandraAlert>, C> getPageAlertsQuery(CassandraAlertFilter filter,
                      SortSpecification<CassandraAlertSortAttribute> sortSpecification,
                      MarkPageRequest<CassandraAlert> pageRequest);

    WriteQuery<CassandraAlert, C> getAddAlertQuery(CassandraAlert alert);
```


This interface has all the queries that are to be used by the demo application. Here is an implementation example of the interface shown above.

The DistQueryManager provides all queries required by the business logic without exposing the underlying generic queries directly. This also helps the application to keep a check on the queries that can be issued to the database. Random queries are not to be accepted. The business logic uses one of the interface API listed in the interface to perform persistence operations at a given point in time. An example is shown below. Earlier examples showed that business logic references distributed data store service and distributed query service. The following example shows how these references are put to use.

CassandraAlertManager.java Posting Alert:

```java
@Override
public CassandraAlert post(Severity severity, CassandraAlertTopic topic, String origin, String data) throws PersistenceException {
    if (topic == null) {
        throw new NullPointerException(...);
    }

    CassandraAlert alert = new CassandraAlert(sysId, true, topic.id(), origin, new Date(), severity, data);
    WriteQuery<CassandraAlert, DataStoreContext> postAlertQuery = queryService.getAddAlertQuery(alert);
    try {
        alert = dataStoreService.execute(postAlertQuery);
    } catch (Exception e) {
        ...
    }
}
```
The method from the previous listing posts a new Alert into the database. It is a write query that creates a new row for every alert posted. The post method is called from other components whenever they want to log an alert message in the database. In this method, the call flow is as follows:

1. Create a transport object (DTO) for the incoming alert
2. Call the Distributed Query Service API (getAddAlertQuery) to get an object of type AddQuery. Please see the implementation above for details. The DTO is an input to this method.
3. Call the Distributed DataStoreService API (execute) to execute the query and pass the postAlertQuery as an argument.
4. Return the stored Alert on success or throw a PersistenceException on a failure.

This sequence is followed for every write query to the persistence layer from business logic.

The following listing illustrates another example of business logic using persistence layer services using a query service. This is a read operation and the example code is as follows.

**CassandraAlertManager.java Reading from the Database:**

```java
@Override
public List<CassandraAlert> find(CassandraAlertFilter alertFilter,
                                 SortSpecification<CassandraAlertSortAttribute> sortSpec) {
    try {
        ReadQuery<List<CassandraAlert>, DataStoreContext> query =
            queryService.getFindAlertsQuery(alertFilter, sortSpec);
        return dataStoreService.execute(query);
    } catch (Exception e) {
        ...
    }
}
```

```java
@Override
public MarkPage<CassandraAlert> find(CassandraAlertFilter alertFilter,
                                      SortSpecification<CassandraAlertSortAttribute> sortSpec,
                                      MarkPageRequest<CassandraAlert> pageRequest) {
    ReadQuery<MarkPage<CassandraAlert>, DataStoreContext> query =
        queryService.getPageAlertsQuery(
            alertFilter, sortSpec, pageRequest);
    try {
        return dataStoreService.execute(query);
    } catch (Exception e) {
        ...
    }
}
```
The two methods shown read from the database in different ways. The first one issues a find query using a filter object. The filter specifies the pivot around which the query results are read. The second method reads a page of alerts and is used when there is a need to paginate results. This is mostly used by a GUI where pages of Alerts are displayed instead of a single long list of Alerts.

The following is an example of filter object as defined in the demo application.

CassandraAlertFilter.java:

```java
cassandraAlertFilter
```

Every application needs to define its filter parameters as in the above code. In the demo application, there is severity filter to “find Alerts where Severity = CRITICAL, WARNING” for example. So, Severity is a Set condition. The find method returns the row if one of the values in a set condition match. The other conditions in the demo follow similar principles.

They cater to various conditional queries that can be issued as a read query to the database. The caller who wants to read from the database needs to create a filter object and fill it with appropriate values before issuing a find query.

**Data Access Object - DAO**

In the previous information, the business logic called the DataStoreService API to perform any persistence operation. The API performs the operation using a DAO. The DAO is a layer that acts as a single point of communication between the business logic and the database. The infrastructure provides generic abstractions of the DAO. However, each table needs to have a table or a Column family specific DAO defined. For this Alerts Demo application there is a CassandraAlertDao. The example code is illustrated in the following listing.

CassandraAlertDao.java:

```java
cassandraAlertDao
```
CassandraAlertSortAttribute> {

public CassandraAlertDao() throws PersistenceConnException {
    cfList.add(new AlertsBySeverity());
    cfList.add(new AlertsByState());
    cfList.add(new AlertsByTopic());
    cfList.add(new AlertsByOrigin());
    cfList.add(new AlertsByTimeStamp());
    cfList.add(new AlertsCount());
    cfList.add(new AlertsByUidAndSysId());
}

private static class AlertColumnFamily {
    private static final ColumnName<String, String> SYS_ID_NAME =
        ColumnName.valueOf("sysId", BasicType.UTF8, false);
    private static final ColumnName<String, Severity> SEVERITY_COL_NAME =
        ColumnName.valueOf("severity", BasicType.UTF8, false);
    private static final ColumnName<String, Date> TIMESTAMP_COL_NAME =
        ColumnName.valueOf("timestamp", BasicType.DATE, false);
    private static final ColumnName<String, String> DESC_COL_NAME =
        ColumnName.valueOf("description", BasicType.UTF8, false);
    private static final ColumnName<String, Boolean> STATE_COL_NAME =
        ColumnName.valueOf("state", BasicType.BOOLEAN, false);
    private static final ColumnName<String, String> ORIGIN_COL_NAME =
        ColumnName.valueOf("origin", BasicType.UTF8, false);
    private static final ColumnName<String, String> TOPIC_COL_NAME =
        ColumnName.valueOf("topic", BasicType.UTF8, false);

    private static final ColumnFamily<String, String> COL_FAMILY =
        ColumnFamily.newColumnFamily("Alerts", StringSerializer .get(),
                    StringSerializer .get(),
                    ByteSerializer .get());

    private static Collection<ColumnName<String, ?>> cfMeta;
    static {
        Collection<ColumnName<String, ?>>tmpCfMeta =
            new ArrayList<ColumnName<String, ?>>();
        tmpCfMeta.add(SYS_ID_NAME);
        tmpCfMeta.add(DESC_COL_NAME);
        tmpCfMeta.add(ORIGIN_COL_NAME);
        tmpCfMeta.add(SEVERITY_COL_NAME);
        tmpCfMeta.add(STATE_COL_NAME);
        tmpCfMeta.add(TIMESTAMP_COL_NAME);
        tmpCfMeta.add(TOPIC_COL_NAME);
        cfMeta = Collections.unmodifiableCollection(tmpCfMeta);
    }

private static ColumnFamily<String, String> COL_FAMILY =
        ColumnFamily.newColumnFamily("Alerts", StringSerializer .get(),
                    StringSerializer .get(),
                    ByteSerializer .get());

private static Collection<ColumnName<String, ?>> cfMeta;
static {
    Collection<ColumnName<String, ?>>tmpCfMeta =
        new ArrayList<ColumnName<String, ?>>();
    tmpCfMeta.add(SYS_ID_NAME);
    tmpCfMeta.add(DESC_COL_NAME);
    tmpCfMeta.add(ORIGIN_COL_NAME);
    tmpCfMeta.add(SEVERITY_COL_NAME);
    tmpCfMeta.add(STATE_COL_NAME);
    tmpCfMeta.add(TIMESTAMP_COL_NAME);
    tmpCfMeta.add(TOPIC_COL_NAME);
    cfMeta = Collections.unmodifiableCollection(tmpCfMeta);
}
private static ColumnFamilyDefinition<String, String> CF_DEF =
    new ColumnFamilyDefinition<String, String>(
        COL_FAMILY, BasicType.UTF8, BasicType.BYTES,
        BasicType.UTF8, null, cfMeta);
private static final EnumColumnDecoder<String, Severity> SEV_DECODER
    = new EnumColumnDecoder<String, Severity>(Severity.class);

private AlertColumnFamily() {
}

public static int compareColumns(
    CassandraStorable<String, String> row1,
    CassandraStorable<String, String> row2,
    CassandraAlertSortAttribute sortBy) {
    int retVal = 0;

    switch (sortBy) {
        case ORIGIN:
            StringColumn<String> col1 =
                (StringColumn) row1.getColumn(ORIGIN_COL_NAME);
            StringColumn<String> col2 =
                (StringColumn) row2.getColumn(ORIGIN_COL_NAME);
            retVal = col1.compareTo(col2);
            break;
        case TIMESTAMP:
            DateColumn<String> time1 =
                (DateColumn) row1.getColumn(TIMESTAMP_COL_NAME);
            DateColumn<String> time2 =
                (DateColumn) row2.getColumn(TIMESTAMP_COL_NAME);
            retVal = time1.compareTo(time2);
            break;
        case SEVERITY:
            EnumColumn<String, Severity> sev1 =
                (EnumColumn) row1.getColumn(SEVERITY_COL_NAME);
            EnumColumn<String, Severity> sev2 =
                (EnumColumn) row2.getColumn(SEVERITY_COL_NAME);
            retVal = sev1.compareTo(sev2);
            break;
        case STATE:
            BooleanColumn<String> state1 =
                (BooleanColumn) row1.getColumn(STATE_COL_NAME);
            BooleanColumn<String> state2 =
                (BooleanColumn) row2.getColumn(STATE_COL_NAME);
            retVal = state1.compareTo(state2);
            break;
    }
    return retVal;
}
case TOPIC:
    StringColumn<String> topic1 =
        (StringColumn) row1.getColumn(TOPIC_COL_NAME);
    StringColumn<String> topic2 =
        (StringColumn) row2.getColumn(TOPIC_COL_NAME);
    retVal = topic1.compareTo(topic2);
    break;
} return retVal;
}

In this code, there is defined a constructor and the main column family. The Alerts in this code is
the main column family in the CassandraAlertDao and has following columns:

- sysId
- severity
- timestamp
- origin
- topic
- description
- state

These columns are defined along with the data type for each column, a decoder to assist in the
read operation and a method to compare columns while sorting a read result.

In addition to this column family, free form queries are supported on a combination of values of
severity, timestamp, origin, topic, state and description.

To enable this, a secondary index for each of these columns needs to be created and maintained.
This secondary index is another column family and it is called the secondary column family. An
example is AlertsBySeverity column family as shown below.

The secondary column families use composite columns and a row in AlertsBySeverity would look
like this.

RowKey ➔ CRITICAL : 1 | CRITICAL : 2 | INFO : 3 | WARNING : 5 | …..

Here the first part of the composite column name is the value of Severity that is wanted to match
and the second part of the column name is the primary key / row key of the matching row in the
main column family. To lookup all Alerts matching Severity = CRITICAL, rows 1 and 2 will be
returned. Do an additional lookup in the main column family to retrieve the data from rows 1 and
2. Once the data is retrieved, convert the same into a storable and return the query back to the
application.

CassandraAlertDao.java AlertsBySeverity Column Family:

```java
public static class SeverityComposite implements Serializable, Comparable<SeverityComposite>{
    @Component (ordinal = 0)
    private String severity;
```
@Component (ordinal = 1)
private String id;

private SeverityComposite(Severity severity, String id) {
    this.severity = (severity == null) ? null : severity.name();
    this.id = id;
}

public Severity getSeverity() {
    return Enum.valueOf(Severity.class, this.severity);
}

public String getId() {
    return this.id;
}

@Override
public int hashCode() {
    ...
}

@Override
public boolean equals(Object obj) {
    ...
}

@Override
public int compareTo(SeverityComposite other) {
    int comparison = 0;
    if (other.id != null) {
        comparison = id.compareTo(other.id);
    }
    if (comparison == 0) {
        comparison = this.severity.compareTo(other.severity);
    }
    return comparison;
}

private static class AlertsBySeverity
    implements CfQueryOperations<String, CassandraAlert> {
    private static final AnnotatedCompositeSerializer<SeverityComposite>
        serializer = new AnnotatedCompositeSerializer<SeverityComposite>
            (SeverityComposite.class);

private static final ColumnFamily<String, SeverityComposite> COL_FAMILY
    = ColumnFamily.newColumnFamily("AlertsBySeverity",
        StringSerializer.get(), serializer, ByteSerializer.get());

private static final ColumnFamilyDefinition<String, SeverityComposite>
    CF_DEF = new ColumnFamilyDefinition<String, SeverityComposite>{
        COL_FAMILY, BasicType.UTF8, BasicType.BYTES,
        new CompositeType(BasicType.UTF8, BasicType.UTF8),
        "Alerts By Severity CF"};

private static final String ROW_KEY = "AlertsBySeverity";
private static final
    Provider<ColumnDecoder<SeverityComposite, ?>,
    ColumnName<SeverityComposite, ?>> SEVERITY_DECODER = new Provider
    <ColumnDecoder<SeverityComposite, ?>,
    ColumnName<SeverityComposite, ?>>() {

        @Override
        public ColumnDecoder<SeverityComposite, ?>
        get(ColumnName<SeverityComposite, ?> entity) {
            return ValuelessColumnDecoder.getInstance();
        }
    };

@Override
public void prepareMutation(CassandraAlert transportable,
    DataStoreContext context) throws Exception {
    CassandraStorable<String, SeverityComposite> storable = new
        CassandraStorable<String, SeverityComposite>(ROW_KEY);
    storable.setColumn(new ValuelessColumn<SeverityComposite>(
        ColumnName.<SeverityComposite, Void> valueOf(
            new SeverityComposite(transportable.getSeverity(),
                transportable.getId().getValue()))));

    context.getContext().prepareMutation(COL_FAMILY, storable);
}

@Override
public void prepareTransaction(CassandraAlert transportable,
    DataStoreContext context) throws Exception {
    context.getTransactionContext()
        .prepareTransaction(COL_FAMILY.getName(), ROW_KEY);
}

99
@Override
public void prepareDelete(CassandraAlert transportable,
        DataStoreContext context) throws Exception {
    SeverityComposite deleteColumn =
            new SeverityComposite(transportable.getSeverity(),
                                 transportable.getId().getValue());
    context.getContext().delete(COL_FAMILY, ROW_KEY,
                         ColumnName
               .<SeverityComposite, Void>
               .valueOf(deleteColumn));
}

@Override
public void prepareUpdate(CassandraAlert oldT, CassandraAlert newT,
        DataStoreContext context) throws Exception {
    ...
}

In this code, SeverityComposite is the object that represents a composite column for AlertsBySeverity. AlertsBySeverity implements CfQueryOperations interface. This interface contains following methods.

1. prepareTransaction–prepares a secondaryColumn family row write transaction.
2. prepareMutation–prepares a secondary Column family row write.
3. prepareDelete–prepares a delete of a secondary index column.
4. prepareUpdate–prepares an update operation on AlertsBySeverity.

When a write query is issued from business logic, a new row is created or an existing row is updated in the main column family.

In addition, there is need to create/update the secondary column families to keep the queries updated. The above mentioned interface operations provide an abstraction to perform a write on all secondary column families along with the main column family.

The secondary column family needs to define the necessary serializers for composite columns and a RowKey. In the demo code, every secondary column family has exactly one very wide row. This is done to achieve faster lookup during a read operation. If the data exceeds the upper limit of the number of columns (2 billion columns), other methods such as sharding can be used to partition the secondary index.

In the example, AlertsBySeverity is shown. Similar code needs to be written for each secondary column family that is needed by the query operations of the application.

Once all secondary column families are defined along with main column family, the DAO needs to provide the following methods. The example code of these methods as defined in the demo application is presented here.

cast()

This method is used during read operations. When a row needs to be returned to the application, it converts the data from storable format to a DTO.
CassandraAlertDao.java:

```java
@Override
public CassandraAlert convert(CassandraStorable<String, String> source) {
    if (source == null) {
        throw new NullPointerException(...);
    }

    final CassandraAlert alert = new CassandraAlert(source.getId());
    ColumnVisitor<String> visitor = new ColumnVisitorAdapter<String>() {
        @Override
        public void visit(BooleanColumn<String> column) {
            alert.setState(column.getValue());
        }

        @Override
        public void visit(DateColumn<String> column) {
            alert.setTimestamp(column.getValue());
        }

        @Override
        public void visit(StringColumn<String> column) {
            if (AlertColumnFamily.DESC_COL_NAME.equals(column.getName())) {
                alert.setDescription(column.getValue());
            } else if (AlertColumnFamily.ORIGIN_COL_NAME.equals(column.getName())) {
                alert.setOrigin(column.getValue());
            } else if (AlertColumnFamily.TOPIC_COL_NAME.equals(column.getName())) {
                alert.setTopicId(column.getValue());
            } else if (AlertColumnFamily.SYS_ID_NAME.equals(column.getName())) {
                alert.setSysId(column.getValue());
            }
        }

        @Override
        public void visit(EnumColumn<String, ? extends Enum<?>> column) {
            if (AlertColumnFamily.SEVERITY_COL_NAME.equals(column.getName())) {
                alert.setSeverity((Severity) column.getValue());
            }
        }
    };

    for(Column<String, ?> col : source.getColumns()) {
        col.accept(visitor);
    }
```
getColumnDecoder()

getColumnDecoder - This method takes a column as argument and returns the data type of that column to the caller. This is required for reading the columns in correct format.

CassandraAlertDao.java:

```java
@Override
protected ColumnDecoder<String, ?> getColumnDecoder(ColumnName<String, ?> columnName) {
    if (columnName == null) {
        throw new NullPointerException(...);
    }

    if (AlertColumnFamily.SEVERITY_COL_NAME.equals(columnName)) {
        return AlertColumnFamily.SEV_DECODER;
    } else if (AlertColumnFamily.TIMESTAMP_COL_NAME.equals(columnName)) {
        return DateColumnDecoder.getInstance();
    } else if (AlertColumnFamily.DESC_COL_NAME.equals(columnName) ||
              AlertColumnFamily.ORIGIN_COL_NAME.equals(columnName) ||
              AlertColumnFamily.TOPIC_COL_NAME.equals(columnName) ||
              AlertColumnFamily.SYS_ID_NAME.equals(columnName)) {
        return StringColumnDecoder.getInstance();
    } else if (AlertColumnFamily.STATE_COL_NAME.equals(columnName)) {
        return BooleanColumnDecoder.getInstance();
    }
    return null;
}
```

createStorableInstance()

This method converts the DTO into a storable format. Storable format is the one which underlying database client code understands. More on this in the next section.

CassandraAlertDao.java:

```java
@Override
protected CassandraStorable<String, String> createStorableInstance(CassandraAlert transportable) {
    CassandraStorable<String, String> storable =
            new CassandraStorable<String, String>(
                    transportable.uid(), transportable.getSysId());

    storable.setColumn(new StringColumn<String>(
            AlertColumnFamily.SYS_ID_NAME, transportable.getSysId()));
    storable.setColumn(new StringColumn<String>(
            AlertColumnFamily.SEVERITY_COL_NAME, transportable.getSeverity()));
    storable.setColumn(new StringColumn<String>(
            AlertColumnFamily.TIMESTAMP_COL_NAME, transportable.getTimestamp()));
    storable.setColumn(new StringColumn<String>(
            AlertColumnFamily.DESC_COL_NAME, transportable.getDescription()));
    storable.setColumn(new StringColumn<String>(
            AlertColumnFamily.ORIGIN_COL_NAME, transportable.getOrigin()));
    storable.setColumn(new StringColumn<String>(
            AlertColumnFamily.TOPIC_COL_NAME, transportable.getTopic()));
    storable痼tColumn(new StringColumn<String>(
            AlertColumnFamily.STATE_COL_NAME, transportable.getState()));

    return storable;
}
```
AlertColumnFamily.DESC_COL_NAME, transportable.getDescription()));
storable.setColumn(new EnumColumn<String, Severity>(
    AlertColumnFamily.SEVERITY_COL_NAME,
    transportable.getSeverity()));
storable.setColumn(new DateColumn<String>(
    AlertColumnFamily.TIMESTAMP_COL_NAME,
    transportable.getTimestamp()));
storable.setColumn(new BooleanColumn<String>(
    AlertColumnFamily.STATE_COL_NAME,
    transportable.getState()));
storable.setColumn(new StringColumn<String>(
    AlertColumnFamily.ORIGIN_COL_NAME, transportable.getOrigin()));
storable.setColumn(new StringColumn<String>(
    AlertColumnFamily.TOPIC_COL_NAME, transportable.getTopicId()));
return storable;
}

conform()

This method is used during an update operation.

CassandraAlertDao.java:

@Override
protected CassandraAlert conform(
    CassandraAlert alert, CassandraAlert alert2) {
    if (alert2 == null) {
        throw new NullPointerException(...);
    }
    if (alert == null) {
        return alert2;
    }

    if (alert.getState() != alert2.getState()) {
        alert.setState(alert2.getState());
    }
    return alert;
}

columnFamilyDefinitions()

The abstraction layer calls this method to perform operations on secondary column families.

CassandraAlertDao.java:

@Override
protected Collection<ColumnFamilyDefinition<?, ?>>
columnFamilyDefinitions() {
    Collection<ColumnFamilyDefinition<?, ?>> colFamilies = new
    ArrayList<ColumnFamilyDefinition<?, ?>>();

colFamilies.add(AlertColumnFamily.CF_DEF);
colFamilies.add(AlertsBySeverity.CF_DEF);
colFamilies.add(AlertsByState.CF_DEF);
colFamilies.add(AlertsByTopic.CF_DEF);
colFamilies.add(AlertsByOrigin.CF_DEF);
colFamilies.add(AlertsCount.CF_DEF);
colFamilies.add(AlertsByUidAndSysId.CF_DEF);
colFamilies.add(AlertsByTimeStamp.CF_DEF);

return colFamilies;
}

getMainColumnFamily()
This method returns a handle to the main column family.
CassandraAlertDao.java
@override
protected ColumnFamilyDefinition<String, String> getMainColumnFamily() {
    return AlertColumnFamily.CF_DEF;
}

findRows()
This method is used to find the row keys that match a specific search criteria. Used during find operations.
The abstraction layer calls this method.
CassandraAlertDao.java:
@override
protected Collection<String> findRows(CassandraAlertFilter filter, final DataStoreContext context) throws PersistenceException, Exception {
    Collection<String> rowsSet = new ArrayList<String>();
    if (filter == null) {
        Collection<String> id = new ArrayList<String>();
        Procedure<CassandraStorable<String, String>> procedure = new
            Procedure<CassandraStorable<String, String>>() {
                @Override
                public CassandraStorable<String, String> execute() throws Exception {
                    return (context.getContext().get(
                        AlertsCount.COL_FAMILY, AlertsCount.COUNT_DECODER,
                        AlertsCount.ROW_KEY));
                }
            };

        104
context.getTransactionContext()
    .prepareTransaction(AlertsCount.COL_FAMILY.getName(),
    AlertsCount.ROW_KEY);
CassandraStorable<String, String> row =
    context.getTransactionContext().
    executeCriticalSection(procedure);
for (Column<String, ?> col : row.getColumns()) {
    id.add(col.getName().getValue());
}
return id;

if (filter.getOriginCondition() != null) {
    final ByteBufferRange range;
    switch (filter.getOriginCondition().getMode()) {
    case EQUAL:
        range = AlertsByOrigin.serializer.buildRange()
            .withPrefix(filter.getOriginCondition()
                .getValue());
        break;
    case STARTS_WITH:
        range = AlertsByOrigin.serializer.buildRange()
            .greaterThanEquals(filter
                .getOriginCondition()
                .getValue())
            .lessThan("~");
        break;
    default:
        range = null;
        break;
    }

    // Find Rows for this filter
    Procedure<CassandraStorable<String, Origin>> procedure =
        new Procedure<CassandraStorable<String, Origin>>() {
            @Override
            public CassandraStorable<String, Origin> execute()
                throws Exception {
                return context.getContext()
                    .get(AlertsByOrigin.COL_FAMILY,
                    AlertsByOrigin.ROW_KEY, range,
                    AlertsByOrigin.ORIGIN_DECODER);
            }
        };
}
context.getTransactionContext()
 .prepareTransaction(AlertsByOrigin.COL_FAMILY.getName(),
                      AlertsByOrigin.ROW_KEY);

CassandraStorable<String, Origin> rows = context
 .getTransactionContext().
 executeCriticalSection(procedure);
Collection<String> id = new ArrayList<String>();
for (Column<Origin, ?> orig : rows.getColumns()) {
    id.add(orig.getName().getValue().getId());
}

// Add row Id's to the final Id set
rowsSet.retainAll(id);
}

// Severity Condition. Only IN is supported for now.
if (filter.getSeverityCondition() != null) {
    switch(filter.getSeverityCondition().getMode()) {
    case IN:
        for (Severity sev : filter.getSeverityCondition().
             getValues()) {
            final ByteBufferRange range = AlertsBySeverity.serializer
             .buildRange().withPrefix(sev.name()).
             greaterThan(" ").lessThanEquals(~");
            Procedure<CassandraStorable<String, SeverityComposite>>
            procedure = new Procedure<CassandraStorable<String,
                                         SeverityComposite>>() {
                @Override
                public CassandraStorable<String,
                                         SeverityComposite> execute()
                throws Exception {
                    return context.getContext() .get(AlertsBySeverity.COL_FAMILY,
                                                    AlertsBySeverity.ROW_KEY, range,
                                                    AlertsBySeverity.SEVERITY_DECODER);
                }
            };
        }
    }

context.getTransactionContext()
 .prepareTransaction(AlertsBySeverity.COL_FAMILY.getName(),
                      AlertsBySeverity.ROW_KEY);

CassandraStorable<String, SeverityComposite> rows =
 context.getTransactionContext().
executeCriticalSection(procedure);
Collection<String> id = new ArrayList<String>();
for (Column<SeverityComposite, ?> sevRow :
    rows.getColumns()) {
    id.add(sevRow.getName().getValue().getId());
}
if (rowsSet.isEmpty()) {
    rowsSet.addAll(id);
} else {
    rowsSet.retainAll(id);
}
}
}

//Topic filter
if (filter.getTopicCondition() != null) {
    final ByteBufferRange range;
    switch (filter.getTopicCondition().getMode()) {
    case EQUAL:
        range = AlertsByOrigin.serializer.buildRange()
            .withPrefix(filter.getTopicCondition() 
            .getValue());
        break;
    case STARTS_WITH:
        range = AlertsByOrigin.serializer.buildRange()
            .greaterThanEquals(filter 
            .getTopicCondition()
                .getValue())
            .lessThan("~");
        break;
    default:
        range = null;
        break;
    }
    // Find Rows for this filter
    Procedure<CassandraStorable<String, Topic>> procedure =
        new Procedure<CassandraStorable<String, Topic>>() {
            @Override
            public CassandraStorable<String, Topic> execute()
                throws Exception {
                return context.getContext()
                    .get(AlertsByTopic.COL_FAMILY,
                        AlertsByTopic.ROW_KEY, range,
                        AlertsByTopic.TOPIC_DECODER);
CassandraStorable<String, Topic> rows = context.getTransactionContext().executeCriticalSection(procedure);

// Add row Id's to the final Id set
Collection<String> id = new ArrayList<String>();
for (Column<Topic, ?> topic : rows.getColumns()) {
    id.add(topic.getName().getValue().getId());
}

if(rowsSet.isEmpty()) {
    rowsSet.addAll(id);
} else {
    rowsSet.retainAll(id);
}

// State Filter
if (filter.getStateCondition() != null) {
    final ByteBufferRange range;
    switch(filter.getStateCondition().getMode()) {
    case EQUAL:
        range = AlertsByState.serializer.buildRange() .withPrefix(filter.getStateCondition().getValue()) .greaterThan(" ").lessThan("~");
        break;
    case UNEQUAL:
        range = AlertsByState.serializer.buildRange() .withPrefix(!filter.getStateCondition().getValue()).greaterThan(" ").lessThanEquals("~");
        break;
    default:
        range = null;
        break;
    }

    // Find Rows for this filter
    Procedure<CassandraStorable<String, StateComposite>> procedure = new Procedure<CassandraStorable<String, StateComposite>>() {
@Override
public CassandraStorable<String, StateComposite> execute() throws Exception {
    return context.getContext().get(AlertsByState.COL_FAMILY, AlertsByState.ROW_KEY, range, AlertsByState.STATE_DECODER);
}

// Start the transaction
context.getTransactionContext().prepareTransaction(AlertsByState.COL_FAMILY.getName(), AlertsByState.ROW_KEY);
CassandraStorable<String, StateComposite> rows = context.getTransactionContext().executeCriticalSection(procedure);
// Add the rows to the row set
Collection<String> id = new ArrayList<String>();
for (Column<StateComposite, ?> state : rows.getColumns()) {
    id.add(state.getName().getValue().getId());
}
// Add row Id's to the final Id set
if (rowsSet.isEmpty()) {
    rowsSet.addAll(id);
} else {
    rowsSet.retainAll(id);
}
return rowsSet;
}

findPagedRows()
Same as the previous one but takes paging into account.
CassandraAlertDao.java:
@

protected <M> MarkPage<String> findPagedRows(CassandraAlertFilter filter, SortSpecification<CassandraAlertSortAttribute> sort, final MarkPageRequest<M> pageRequest, final DataStoreContext context) {
    if (filter == null) {
        if (pageRequest == null) {
            throw new RuntimeException("Page request cannot be null");
        }
    }
    return null;
}
// Convert the pageRequest
CassandraStorable<String, String> convertMark =
    (CassandraStorable<String, String>) pageRequest.getMark();
final MarkPageRequest<String> convertedPageRequest =
    pageRequest.convert((convertMark != null)
        ? convertMark.getId() : null);
Procedure<MarkPage<Column<String, ?>> procedure =
    new Procedure<MarkPage<Column<String, ?>>>() {
        @Override
        public MarkPage<Column<String, ?>> execute() throws Exception {
            return context.getContext().get(AlertsCount.COL_FAMILY,
                AlertsCount.ROW_KEY,
                convertedPageRequest,
                AlertsCount.COUNT_DECODER);
        }
    };
try {
    context.getTransactionContext()
        .prepareTransaction(AlertsCount.COL_FAMILY.getName(),
            AlertsCount.ROW_KEY);
} catch (PersistenceException e) {
    throw new RuntimeException(e);
}
MarkPage<Column<String, ?>> result = null;
try {
    result = context
        .getTransactionContext()
        .executeCriticalSection(procedure);
} catch (Exception e) {
    throw new PersistenceException(e);
}

// Get the list of Ids from the page
List<String> id = new ArrayList<String>();
for (Column<String, ?> c : result.getData()) {
    id.add(c.getName().getValue());
}

MarkPageRequest<String> pageRequest1 =
    result.getRequest().convert(result
        .getRequest()
        .getMark()
        .getName())
compareRows()

Compares two rows. This method is used for sorting the result set.

CassandraAlertDao.java:

```java
@Override
protected int compareRows(CassandraStorable<String, String> row1,
    CassandraStorable<String, String> row2,
    SortSpecification
    .SortComponent<CassandraAlertSortAttribute> s) {
    if (s.getSortOrder() == SortOrder.ASCENDING) {
      return AlertColumnFamily.compareColumns(row1, row2, s.getSortBy());
    } else {
      return AlertColumnFamily.compareColumns(row2, row1, s.getSortBy());
    }
}
```

Storable

A storable is a format of data that the south bound side of the persistence layer operates on.

Every DTO is converted to a storable on its journey to the database and gets converted back to DTO on its way back to the application. We define a generic storable called CassandraStorable and is used by all DAOs for all DTOs.

The convert routine has been described in the previous section. The CassandraStorable stores data in the form of a map very similar to the underlying database. The application only uses the storable and need not write one for itself.

Backup and Restore

The SDN controller provides a framework to backup and restore controller and application state in a backup file. Backup operation starts with the admin issuing a REST command to backup the controller and applications. Once the operation is completed, the backup file can be copied and stored for later use. In the event of a disaster recovery situation, the stored backup file can be uploaded and restored via REST API. This section provides a brief description for the application developers to enable backup/restore functionality on the new applications.

Backup

A controller backup takes a snapshot of the controller state, and includes the following in a single file:
• Controller databases
• License compliance history and metrics log data
• In a teaming environment, the teaming configuration
• User repository folder (for user-installed applications)
• Controller configuration folder

All application data that goes into the controller databases will be automatically backed up. The applications should consider using backup and restore if they have external data that is not already a part of the list mentioned above.

Any application that needs to backup specific data needs to register with BackupRestoreService and implement BackupRestoreListener interface. This will ensure that the applications get a callback to stage their backup to a specific directory provided by the BackupRestoreService. The applications can control any data that goes into the specified directory, but the directory itself is controlled by the backupRestoreService. The applications will not be able to create or delete the specified directory. When the backup operation is complete, applications will get another callback indicating a backup is complete. Once the application has staged its back up, it could wait for backup to complete or resume its operations; this decision is implementation dependent.

**Restore**

The restore operation is triggered during disaster recovery. The admin uploads the backup file and issues a REST call to restore. The restore process takes place in two steps. In the first step, the applications themselves are restored on the controller as part of the controller restore. Once this step is complete and the restored applications have registered with the backupRestoreService, the second step is triggered. In the second step, the applications get a call back to restore their specific data and state that were not a part of the controller structures listed in the previous section.

**Example:**

```java
@Component
@Service
public class SampleBackupAwareApplication implements BackupRestoreListener {
    @Reference(cardinality = ReferenceCardinality.MANDATORY_UNARY,
               policy = ReferencePolicy.DYNAMIC)
    private volatile BackupRestoreService backupRestoreService;

    @Activate
    protected void activate() {
        backupRestoreService.register(this);
    }

    @Override
    public void onBackupStart(Path directory) {
        try {
            // Stage application specific backup in the specified directory
        }
    }
}
```
// Please make sure you catch exceptions and throw runtime exceptions
// so that backup service can operate correctly.
// If the error is not thrown, backup service will assume
// a successful staging operation for the given application.
} catch (Exception e) {
    // Throw RuntimeException if you need to stop backup in the
    // event of a failure.
    throw new RuntimeException (e);
}

@Override
public void onBackupDone(BackupRestoreStatus status) {
    // Take application specific action on backup done.
    // The status can be SUCCESS or FAILURE
    // The behavior of the application on backup done is implementation
dependent
}

@Override
public void onRestoreStart(Path directory) {
    try {
        // Restore application specific data from the specified directory
        // Please catch exceptions and throw runtime exceptions
        // so that restore service can sense failures.
        // If the error is not thrown, restore service will assume
        // a successful staging operation for the given application.
    } catch (Exception e) {
        // Throw RuntimeException if you need to stop restore in the
        // event of a failure.
        throw new RuntimeException (e);
    }
}

@Override
public void onRestoreDone(BackupRestoreStatus status) {
    // Application specific behavior on Restore done event.
}

Device Driver Framework

Device Driver Framework Overview

The SDN Controller provides a Device Driver Framework with the following capabilities:

- Maintains identity information about the types of physical devices recognized by the framework.
- Determines the type of each physical device using information discovered through the OpenFlow handshake, as well as direct interaction with the device.
- Communicates with the physical device directly to extract configuration information and adjust its type if necessary.
- Persists the discovered device and its configuration information as well as its interface list. For OpenFlow devices, the interface information is reported via the OpenFlow handshake. For non-OpenFlow devices, software known as a Handler Facet are used to interact with the device to obtain the interface list.
- Allows device-specific software components to be written to interact with devices (known as Facets and Handler Facets); these software implementations are associated with a device type.
- Maintains security credentials to allow interaction with devices using protocols such as SNMP and NetConf.

Each of these capabilities is discussed in more detail below.

Facets and Handler Facets

One of the primary reasons for the Device Driver Framework is to allow software components to be developed that can interact with a device. In order to interact with a device, the software component must know the capabilities and characteristics of the device. These software components are referred to as “Facets” and “Handler Facets”. Below are the definitions for a Facet and Handler Facet.

**Facet:** Software that is used to perform a function that does not require direct interaction with the device, but requires knowledge of the device’s capabilities and limitations.

**Handler Facet:** Software that is used to perform a function that requires interaction with a device.

Note: although there is a difference between Facets and Handler Facets, the term Facet is used throughout this section to refer to either one.

Facets are written to access a particular attribute or feature on a device and therefore are tied to specific device types. For example, there may be different Facet implementations for configuring VLANs on HP devices and Cisco devices. They may be different Facet implementations even though they perform the same function (configuring VLANs). Device information stored in XML files (see below) will indicate that it supports the Vlan Facet, but when the Vlan Facet is accessed, the type of device will determine which implementation is used. In this way, the application or
user of the device driver framework does not require knowledge of the type of device with which it is interacting.

**Device Type Information**

Information describing a type of device is stored in XML files. This information describes the attributes (capabilities) of a type of device, not an actual device that exists on the network. As physical devices are discovered, the Device Driver Framework will determine the best device type for the discovered device.

Device type information is stored in XML files and organized in a hierarchical fashion with more specific types extending from more general device types. The following figure illustrates this concept.

At the top of the tree is the BaseSwitch.xml file. This XML file defines a “Default Switch”. The default switch defines only a DeviceIdentity Facet that can be used to get basic information about the device. This indicates that it has a DeviceIdentity Facet available to be accessed; the implementation of the DeviceIdentity Facet will be a specific class name to get instantiated when the Facet is accessed.

Extending from the BaseSwitch are XML files that contain more information about a type of device. For example, at this level the XML files contain information about a “generic” HP device, or some other generic device. The HP.xml file specifies the vendor as HP, and it defines several Facets that can be used with all HP devices. One example is the DeviceIdentityHandler Facet that can be used to interact with the device via SNMP to determine more granular information. For example, for HP devices the model, serial number, specific flags such as type of chassis can be obtained to better determine the specific type of HP device.

At the next level, the figure shows XML files for several HP devices. The 5400.xml file specifies the 5400 device type and it extends from the HP Switch type. This file also defines device types for
each product in the 5400 product line. The device type J9642A is the 6 slot 5406zl product, and the J9643A is the 12 slot 5412zl product. For each specific device type the XML file contains the product description, the SNMP sysObjectld assigned to the device, and Facets that can be used with this type of device. The XML file may also contain “flags” that define additional information about the type of device. For example, the 5400.xml file contains a flag indicating 5400 products are chassis products. This information can be used by Facets to allow them to gather additional information about chassis product.

A key point is that XML files determine what Facets can be used for a specific device type. The XML files for each level specify what Facets each device type supports. These are interfaces only so the Applications that need access to a specific device feature will only be using devices that they know support that feature. For each Facet interface in the XML file, there is also a class name listed which is the implementation of that Facet for that specific device type. So the 3500.xml and the 3800.xml both list the FlowMod facet as supported, but each of them gives a different class name for their implementation. These classes implement the FlowMod Facet in a way that’s specific for each device type. For example, when working on a 3500, the 3500’s implementation of the FlowMod Facet is instantiated when the Facet is accessed by an Application.

Below are parts of the XML files for the 3500 and 3800. Highlighted in red is the Facet Interface (Facet name) which is the same for both devices. Highlighted in blue is the Facet class which is different for the two devices.

3500.xml File

```
<deviceDriver description="HP 3500 Switch">

  <type name="3500" extends="HP Switch">
    <facet name="com.hp.sdn.dvc.facet.FlowModFacet"
           class="com.hp.sdn.dvc.facet.impl.FlowModProVision"/>

    <family>ProCurve 3500</family>
    <product>3500</product>
  </type>

  <type name="J8692A" extends="3500" description="Switch 3500yl-24G">
    <oid>.1.3.6.1.4.1.11.2.3.7.11.58</oid>
    <model>J8692A</model>
    <product>Switch 3500yl-24G</product>
  </type>

</deviceDriver>
```

3800.xml File

```
<deviceDriver description="HP 3800 Switch">

  <type name="3800" extends="HP Switch">
    <facet name="com.hp.sdn.dvc.facet.FlowModFacet"
           class="com.hp.sdn.dvc.facet.impl.FlowModChassisV2"/>

</deviceDriver>
```
Component Responsibilities

The following figure illustrates the components that make up the Device Driver Framework.

**Device Type XML Files:** The XML files contain information describing the attributes and supported Facets for a device type.

**Device Driver Manager:** The Device Driver Manager loads the information from the XML files and creates an in-memory representation of the Device Type information. The Device Driver Manager assists Discovery components to determine the best Device Type for discovered devices.

**Device Discovery:** The figure illustrates that there can be many discovery components. The OpenFlow (OF) discovery component is shown in the figure. OF Device Discovery will be notified by the Controller Service when an OpenFlow device connects to the controller. OF Device Discovery is then responsible to determine the type of device based on information provided in the OpenFlow handshake and the XML information maintained by the Device Driver Manager.
Handler Facet may also be used to obtain information directly from the device in order to better determine its type.

**Device Manager:** The Device Manager receives information about devices from the discovery components. The Device Manager will maintain the device information, store the information in a database, and share the information with other team members if configured to operate as a member of a team.

**Key Manager:** The Key Manager is responsible for storing security information (referred to as keys) that is required to interact with a device using protocols such as SNMP or NetConf. The network administrator is responsible for loading keys into the Key Manager through its REST API (refer to the REST API specification for more information). The Key Manager is responsible for storing the keys in a local database and sharing them with other SDN Controller team members. The Key Manager will provide keys to Applications and Facets that want to interact with devices.

**Handler Facet:** Handler Facets are used to interact with a device to obtain or modify device information using protocols such as SNMP to communicate with the device. For example, the HpSnmpDeviceIdentity class is an implementation of a Handler Facet. It uses SNMP to read the sysObjectId of the discovered device. The sysObjectId is the vendor’s authoritative identification of the device. If a matching sysObjectId can be found in the Device Type information (information in XML files), then the type of device can be determined.

**Applications:** Applications can use the Device Service to retrieve Device objects through which it can obtain Facets that work with the device. It is not shown in the figure, but applications can use KeyService APIs to perform Create, Read, and Delete operations on device keys, and DeviceDriverService APIs to determine the device type.

---

**Example Operation**

The following example is provided to illustrate how the Device Driver Framework works. The steps are numbered and the numbers correspond to the interactions shown in the figure above.

1) When the SDN Controller starts, the Device Driver Manager will load the XML information (which is stored in resource bundles) and create an in-memory representation of the Device Type information.

2) When an OpenFlow device connects to the SDN Controller, the Controller Service will notify the OF Device Discovery component. The Controller Service will provide information discovered about the device as part of the OpenFlow session establishment process. This information consists of manufacture description, hardware description, software description, serial number, and Data Path ID (dpid).

3) Using the OpenFlow information, OF Device Discovery will interact with the Device Driver Manager to attempt to determine the device’s type. The OpenFlow information may be adequate to determine the exact type of device. For example, if the hardware description matches product information in the Device Type data (i.e., XML data), then the exact type of device can be determined. However, if the OpenFlow information is not adequate to determine the exact type of device, then further discovery is required. This will require interacting with the device to obtain information needed to determine its type.
4) Assume that OF Device Discovery was unable to determine the exact type of device in step 3. Also assume in step 3 that it is determined that the device is manufactured by HP. OF Device Discovery will interact with the Device Driver Manager to obtain the HP Device Identity Handler Facet that can be used to interact with the device using SNMP to obtain additional information.

5) The HP Device Identity Handler Facet must obtain the correct security key to interact with the device. The Device Identity Handler Facet will obtain all SNMP security keys from the Key Manager. It will try keys until it finds a key that allows it to get (read) SNMP objects from the device. Note, trying all keys is only required the first time a Device Identity Handler Facet tries to interact with a device. Once the correct key is determined, it is saved and will be used in subsequent interactions with the device.

6) Once the correct SNMP key is discovered, the Facet will get several SNMP MIB objects from the device to enable it to better identify the type of device. The sysObjectld is the MIB object that will be used to determine the type of device.

7) OF Device Discovery will use this additional information and interact with the Device Driver Manager to better determine the type of this device. The device type will be determined using the sysObjectld and looking for a match in the Device Type data (i.e., XML data).

8) The type of device as well as other information discovered about the device (through SNMP or through the OpenFlow information) is packaged in a DeviceInfo object. OF Device Discovery will pass the DeviceInfo object to the Device Manager. The Device Manager will store or update the Device information in its database, and share it with other team members if configured to operate in a team.

9) Applications can use Device Service APIs to read device information and its associated attributes.

Port-Interface Discovery
The discussion above focused on discovering devices and maintaining information about devices. The Device Manager is also responsible for maintaining information about port-interfaces associated with a device. It is the responsibility of the Discovery components to obtain the port-interface information and provide it to the Device Manager.

How the port-interface information is discovered and kept up to date is the responsibility of the Discovery component. Different techniques will be used depending on the type of Discovery component. For example, OF Device Discovery will obtain port-interface information using the Controller Service API. When a new OpenFlow device is discovered, OF Device Discovery will obtain the port-interface information from the Controller Service. It will also register with the Controller Service to receive notifications when a port-interface is added, deleted, or the port-interface’s status changes.

For non-OpenFlow devices, the interface information can be collected by a Facet using SNMP. Any status changes on the ports discovered can be notified through SNMP Traps if the controller is enhanced to be an SNMP Receiver.
Chassis Devices

A “Chassis Device” is a modular chassis product that accommodates switching and management modules. Modules with different capabilities can be inserted into the chassis. The HP Device Identity Handler Facet will examine the flags in the Device Type data (i.e., XML data) to determine if the device is a chassis product. For HP chassis switches, module identification is important. However, the ability to tell whether the switch is in v1 or v2 mode can only be determined if SNMP is enabled and the Device Identify Facet has the proper key. The module identification affects only the FlowMod Facet. If the chassis is in v2 mode, there is a different implementation of this Facet than the default v1 implementation. v2 modules support Openflow operations in the hardware table that v1 modules do not. Therefore, if the Device Identity Facet cannot recognize the module configuration (because SNMP is not enabled) then it assumes v1 and the Openflow tables are simply less efficient. However, because the module information is not tied to a device type, but rather to a configuration option on a switch of the given type, the FlowMod Facet implementation cannot be specified in the device type’s XML file. The way to circumvent that is to specify a Factory class for the FlowMod Facet implementation and have the Factory query the configuration retrieved from the Device Identity Facet.

Device Objects

Several java objects are used to store information about devices. The following figure illustrates the java objects used to store device information and how the objects are organized.

DeviceType: This object contains information about a device type. It contains all Facets and Handler Facets that are supported by this type of device.

DeviceInfo: This object contains a DeviceType object and other information about an actual device and its configuration.

KeyDeviceInfo: This object extends a DeviceInfo object and contains the security key that is needed to communicate with the device. This object is created when the correct security key is discovered.
**Device:** This object is maintained by Device Manager and contains a DeviceInfo object and other information and status about the device. This object can be obtained by Applications using the Device Service API.

**DeviceHandler:** This object contains a DeviceInfo object and the IP address of the device. This object is used by Application to get Handler Facets.

Note: Ports/Interfaces are maintained as a separate entity associated to a device and are not shown in the figure above.

### Using the Device Driver Framework

Several Device Driver Framework components expose APIs that can be used by Applications. The following APIs will typically be used by Applications:

- **Device Service:** Applications can use APIs provided by this service to perform CRUD (Create, Read, Update, and Delete) operations on devices and port-interfaces. Given a device object, applications can obtain information about Facets, DeviceInfo and Device Handler objects.

- **Key Service:** This Service API allows Applications to add keys, remove keys, and get security keys maintained by the Key Manager.

- **Device Drive Service:** This Service API allows Applications to create Device Handlers. Device Handlers are needed to be able to get Handler Facets that are supported by a specific type of device. Using a Device Handler to get Handler Facets is discussed below.

### Device Model APIs

The java objects that represent Device information expose interfaces to allow Applications to perform device related functions. For example, the Device object provides the method isOnline() to determine if the device is online or offline. The Device object also provides the method info() to obtain a DeviceInfo object. The DeviceInfo object provides an API to allow Application to list of all Facets supported by the device, check if a Facet is supported, and get an instance of a Facet so that the Facet can be used.

### Facet Usage Example

The code sample below demonstrates the following:
- How to get a Device object using the Device Service
- How to get a DeviceInfo object using the Device object
- How to check if a device is online
- How to check if a Facet is supported by the device
- How to get a Facet that is supported by the device
- How to use the Facet to perform a function that is provided by the Facet

```java
/**
 * Validate and adjust the FlowMod passed to this method.
 *
 * @param ds reference to the DeviceService
 * @param dpid DataPathId for the device the FlowMod is to be sent to.
 */
```
private Set<OfmFlowMod> adjustFlowMod(DeviceService ds, DataPathId dpid, OfmFlowMod ofm) {
    Set<OfmFlowMod> adjustedFlows = new HashSet<>(OfmFlowMod);
    try {
        // get the device object associated with the data path ID (dpid)
        Device dev = ds.getDevice(dpid);

        if (!dev.isOnline()) {
            // get the DeviceInfo object which is needed to get a Facet
            DeviceInfo di = dev.info();

            // check if the FlowMod Facet is supported for this device
            if (di.isSupported(FlowModFacet.class)) {
                // get the FlowMod Facet
                FlowModFacet facet = di.getFacet(FlowModFacet.class);

                // call the FlowMod Facet to validate and adjust the FlowMod
                adjustedFlows = facet.adjustFlowMod(ofm);
            }
        }
    }
    catch (Exception e) {
        e.printStackTrace();
        return adjustedFlows;
    }
}

This method demonstrated how to get a Facet and use the Facet. This method is used to validate and adjust, if necessary, FlowMods for a specific device. This method is passed 3 parameters.

- **ds**: A reference to the Device Service. The Device Service is used to get the Device object that contains information about the device.

- **dpid**: This parameter contains the OpenFlow Data Path ID for the device that a FlowMod is to be sent to.

- **Ofm**: This parameter is the FlowMod that is to be validated and adjusted if necessary. NOTE: Applications should not set the table id of the OfmFlowMod. Instead, the FlowModFacet will choose the best table based on the capabilities of the device to which the OfmFlowMod is intended. If the FlowModFacet receives an OfmFlowMod that already has a table id set, it will not adjust the table id for the intended device.

This method will get the FlowMod Facet and use it to validate the FlowMod, and possibly adjust the FlowMod. This method returns a Set of FlowMods. A Set is necessary because it is possible that the device cannot support all the features specified in the original FlowMod in a single table, and several FlowMods are required to achieve the desired behavior.
At line 15 the reference to the Device Service (ds) is used to get a Device object for the Data Path ID. The Device object contains information about the device including the Facets that can be used with that device.

At line 17 the Device object (dev) is used to check if the device is online. If the device is offline, then no validation and adjustment is performed.

At line 19 the Device object is used to get a DeviceInfo object (di). The DeviceInfo object is needed to get Facets that can be used with the device.

At line 22 the DeviceInfo object is used to determine if a Facet called FlowModFacet is supported for the device.

At line 24 the DeviceInfo object is used to get the Facet called FlowModFacet.

At line 27 the Facet is used. The Facet’s adjustFlowMod method is called to validate and adjust a FlowMod.

**Handler Facet Usage Example**

The previous example demonstrated how to get and use a Facet. This example will demonstrate how to get and use a Handler Facet. As discussed above, a Facet does not directly interact with a device, whereas a Handler Facet does. One additional step is required to get a Handler Facet. A Device Handler object is needed in order to get a Handler Facet.

```java
private Set<VlanInfo> getVlans(DeviceService ds, DeviceDriverService dds, DataPathId dpid) {
  Set<VlanInfo> vlans = new HashSet<>(VlanInfo);
  try {
    Device dev = ds.getDevice(dpid);
    DeviceInfo di = dev.info(); // get the DeviceInfo object
    DeviceHandler dh = dds.create(di, getIp(di)); // create a Device Handler
    // check if the Handler Facet is available for the device
```

```java
  return vlans;
}
```
if (dh.isSupported(VlanHandlerFacet.class)) {
    // use the Devcie Handler to get the Handler Facet
    VlanHandler hf = dh.getFacet(VlanHandlerFacet.class);

    // use the Handler Facet to get vlan information
    vlans = hf.getVlans();
}
} catch (Exception e) {
    e.printStackTrace();
}
return vlans;

/**
 * Returns the IP address associated with the DeviceInfo object.
 *
 * @param di DeviceInfo object
 * @return IP address for the DeviceInfo object
 */
private IpAddress getIp(DeviceInfo di) {
    DeviceIdentity facet = di.getFacet(com.hp.device.DeviceIdentity.class);
    return facet.getIpAddress();
}

The method above demonstrated how to get a Handler Facet and use the Handler Facet. This method is used to obtain information about vlans that have been configured on a device. This method is passed 3 parameters.

- **ds**: A reference to the Device Service. The Device Service is used to get the Device object that contains information about a device.
- **ddm**: A reference to the Device Driver Service. The Device Driver Service is needed to allocate a Device Handler. A Device Handler is needed to get a Handler Facet.
- **dpid**: This parameter contains the OpenFlow Data Path ID for the device. This is the device that vlan information will be read from.

This method will return a Set of VlanInfo objects for the specified device. Each VlanInfo object contains information about one vlan.

At line 15 the reference to the Device Service (ds) is used to get a Device object (dev) for the Data Path ID.

At line 17 the Device object is used to get a DeviceInfo object (di). The DeviceInfo object is needed to get a Device Handler and to get the IP address of the device. Both are needed to get a Handler Facet.

At line 19 the Device Driver Service (dds) is used to get a Device Handler that is associated with the DeviceInfo object. The device’s IP address is needed to create a Device Handler. The local
method getIp() is used to get the IP address. The getIp() method is shown in lines 35 through 44, and will use the Device Identity Facet to get the IP address.

At line 22 the Device Handler (dh) is used to determine if the VlanHandlerFacet is available for this device.

At line 24 the Device Handler is used to get the Handler Facet (VlanHandlerFacet).

At line 27 the Handler Facet (hf) is used to interact with the device and retrieve the vlan information.
4 Application Security

Introduction

This chapter provides recommendations and requirements for designing secure applications.

SDN Application Layer

Applications can be implemented in different permutations and combinations of physical and logical instantiations as listed below:

- SDN application inside OSGI container on same operating environment as SDN [“internal” application]
- SDN application via REST interface on same physical HW as SDN [“local external” application]
- SDN application via REST interface on external HW (in single and Distributed Coordination modes) [“remote external” application]
- SDN application running on external cluster of servers but presented as a single instance to a SDN controller

The relevant security components and interfaces generally associated with applications include the following:

- Installation and upgrade authentication (software signatures and validation)
- Application management interface security requirements
- User authentication, including password requirements
- Secure application initialization
- Application to controller mutual authentication
- App Policy enforcement (authorization), including app arbitration, prioritization or hierarchy
- Application high availability features including secure replication
- Secure backup of application data
- REST interface security requirements (such as TLS configuration)
- Application command traceability (identify source of cmds for debugging and security logging)
- Syslog (a computer message logging standard), SNMP notifications and traps, time and clock synchronization

Application Security

Security capabilities are intended to be compatible with NIST SP800-53 Rev 4, typically at the “Moderate Impact System” level except where customer requirements include High Impact or Enhanced Assurance controls. Refer to “Control: The information system” items in section F of the document for the requirements specific to the Moderate Impact classification.

Known requirements for FIPS 140, DoD JITC and Common Criteria should all be applied.
Assumptions

Software development practices

It is assumed that secure and high-assurance development practices are used, including:

- Good design practice including design reviews and threat analysis
- Security awareness through requirements and training
- Static code analysis with corrective action taken before product release
- Product testing includes “negative” testing, i.e., responses to input errors, network protocol fuzzing, etc. are handled in secure and robust manner

Physical security (standalone SDN apps running on servers)

This section applies to instantiations of SDN applications running on “independent Hardware” (e.g., remote external applications). In these cases, physical security is assumed such that only authorized personnel have access to the application host machine.

Logical security (external SDN apps)

To allow for multiple deployment scenarios, we need to assume that communication between the SDN application and the controller is in-band. For external apps, do not assume that an SDN app is connected to the SDN controller by means of a private VLAN. All facets of providing confidentiality, integrity (both system and data), and availability by design therefore apply. Given the nature of an SDN controller interacting with devices, non-repudiation (accountability) is probably also a concern.

Distributed Coordination and Uptime

Any loss of access to the controller might disrupt or otherwise cause loss of network availability to the customer’s network. All configuration, upgrade and maintenance operations, including credentials refresh, must be designed to permit continued controller access during and after these procedures. A cluster/team shutdown must not be required. Inter-controller communications must be authenticated and encrypted using user-supplied credentials.

Secure Configuration

Image validation

The following requirements and guidelines are intended to improve assurance of integrity and interoperability (correct operation):

- The user must be provided with an inventory of the AS-TESTED implementation of the system, including version information for all open source libraries and SHA-1 hashes of all installed files. This information is to be available separately, even if loaded as part of the system installation.
- It is RECOMMENDED that files are distributed to the user such that installation is performed entirely from signed files, the expanded contents of which can be checked against a provided hash. This protects the user from inadvertently installing a version of
an untested or corrupted module. (This is currently expected to be a future REQUIREMENT.)

- External applications performing signature validation (e.g., on updates) SHOULD run with low privilege but require high user privilege (e.g., root) to initiate installation or modification.

**Keys and credentials**

There SHALL NOT be any default credentials. There SHALL NOT be any permanent credentials. Keys used for management and authentication must not be transferable. Keys are to be generated on the device and cannot be injected (configured) from another source. The private key must not be transferable off the device, including configuration backup (Reinstallation requires new credentials).

**File/Encryption requirements**

- Transfer of files to or from the system (once operational) SHOULD (future must) be over a secure transport using FIPS 140 approved algorithms.
- Access to all keys must be password protected. Password based keys must be generated using NIST approved methods.
- All backup and restore operations must be logged, including the identity of the user performing the action.

**Management Interfaces**

**OF interface**

An SDN application must not expose or present a OF interface.

**SSH security**

An SDN application can present a CLI via SSH for configuration and management of the application.

**WebUI**

An SDN application can present its own web UI to configuration application policy and provide status via a web browser. When a web UI is present, the follow requirements exist:

- HTTPS must be available.
- The device must be capable of configuring HTTPS certificates over the HTTPS interface. To prevent a chicken and egg problem, initial configuration of Trust Anchor credentials must be performed through CLI or HTTP.
- Basic Auth must not be used (i.e., no user or system data in URL’s).
- All Open Web Application Security Project (OWASP) security recommendations must be followed.

**Southbound interface**

An SDN application must not interact directly with a managed device. All device communication must be through the controller.
System Integrity

External applications must run in separate memory spaces.

Software validation

- All downloadable files must be signed.
- All file signatures must be validated 1) at time of file saving and 2) loading.
- Signatures and validation shall apply to script files, e.g., Tcl, Python, as well as to binary executables and Java .jar files.
- It is highly desirable to validate system integrity on a running system—boot time is good, but might not be sufficient.

Secure Upgrade

- Updates and configuration changes are to be performed only with sufficient administrative privilege
- Updates must be logged, including both successful and non-successful attempts
5 Including Debian Packages with Applications

This chapter documents the requirements for installing a Debian package with an application. Steps for removing a Debian package are also provided.

Required Services

AppService

The AppService provides:

- Reference to the directory where the contents of the application zip file is extracted
- Current state information for the application
- Ability to register a listener to hear of application specific events

AdminRest

The AdminRest provides:

- Ability to upload a Debian package to the controller
- Ability to run an installation on a Debian package
- Ability to run a removal of a Debian package

Application zip file

When the application is installed via the application manager the contents of the application zip file are extracted to an “unzipped” directory for that application. The application specific directory is located at “/opt/sdn/config/apps/app_id/app_version/unzipped”. All files found within the application zip will be extracted to this location. The extraction process does not perform any validation on the extracted files.

The AppService provides a call to retrieve the Java File object representing this unzipped directory. The required parameters for this call are the application id (string) and the application version (string). The call will return a valid File object that is a directory if the unzipped directory exists for the application id and version, otherwise it throws a not found exception.

An application can use this call to obtain the parent directory for any Debian files that are included with the application:

```java
private static final String APP_ID = "com.hp.sdn.demo.debinst";
private static final String APP_VERSION = "1.0.0.SNAPSHOT";
```
private static final String APP_DEBIAN_FILE = "debinst-sample_1.0_amd64.deb";

private void installDebian() throws IOException {
    File unzipDir = appService.getAppUnzipDir(APP_ID, APP_VERSION);
    File deb = new File(unzipDir, APP_DEBIAN_FILE);

Programming Your Application to Install a Debian Package on the Controller

Determining when to install the Debian Package

The application will need to determine its state at start up to know when it should install its debian packages. When the application is first installed, the state of the application will be INSTALLING. Subsequent normal restarts (such as a restart of the controller) of the application will present a state of ACTIVE. If an application has been disabled and is then subsequently enabled by the user the state of ENABLING will be presented. If an application is upgrading, the first start of the upgraded application will present a state of UPGRADING. Any other state at start of the application indicates some sort of error condition which an application might or might not be able to handle. An application must be able to handle each state. For example, if you program the application to remove the debian package when a user disables the application, you must also program the application to reinstall the debian package when the application is enabled. When the application is started normally (the ACTIVE state), the application should not attempt to install the application, but should decide if there is work it needs to do to determine that the debian package is still installed / running.

An application uses its state to determine when to install a debian package. To install a debian package, the application must be in either the INSTALLING state or the ENABLING state. If an application is in an state other than INSTALLING or ENABLING, the application cannot install the debian package.

The AppService is used to determine the current state of the application. The application component that manages the external debian package must obtain a reference to the AppService. You can obtain a reference to the AppService using annotations and Declarative Services, or by using any other OSGi method. After the AppService is obtained and the current state of the application is determined, the application can use this information to determine whether or not to install the debian package.

The application must handle the states listed in the following switch statement code example:

```
// for application management
@Reference(policy = ReferencePolicy.DYNAMIC,
        cardinality = ReferenceCardinality.MANDATORY_UNARY)
private volatile AppService appService;
```
State state = appService.state(APP_ID);

switch (state) {
    case INSTALLING:
    case ENABLING:
        installDebian();
        installMonitorTask = taskExecutorService.scheduleWithFixedDelay(new 
            InstallMonitor(),
            Measure.valueOf(61, SI.SECOND),
            Measure.valueOf(60, SI.SECOND));
        break;

    case UPGRADING:
        upgradeDebian();
        break;

    case RESOLVED:
    case ACTIVE:
        validateDebian();
        break;

    // should not be in these states at start up
    case NEW:
    case STAGED:
    case UPGRADE_STAGED:
    case CANCELING:
    case DISABLING:
    case DISABLED:
    case UNINSTALLING:
        throw new Exception(E_START_STATE + state);
    }

### AdminRest Interactions

You use the AdminRest class to interact with the AdminResource to upload and install a debian package on the local server (loopback address). Do not use the AdminResource for any other purpose.

The debian file must first be uploaded to the admin process space. The business logic of the AdminRest API copies the debian file delivered with the application zip (from the /opt/sdn/config/apps/<app id>/<app version>/unzipped directory) to the admin upload directory (var/lib/sdn/uploads). This call will occur synchronously, and throws an I/O exception.
if an error occurs writing the file (which will currently manifest itself as an internal server error 500 back to the application code).

After the debian file has been uploaded to the admin space it must be installed. The installation is accomplished via a script that is executed by the business logic in the admin space. The business logic will only look in the admin upload directory for the debian package to install. This is not a synchronous operation in that the underlying script schedules an operation to occur in the future and then returns. From the applications perspective, the call to install a debian will return immediate, but the actual effort to install the debian package will not occur until some point in the future. To install a debian package, execute the following command, where $1 is the name of the debian file:

```
echo "dpkg -i $1 >> /var/log/sdn/admin/install.log 2>&1" | sudo at now + 1 min
```

**Uploading a debian package**

The path for a debian package upload is “/upload”. The REST API requires that a header be provided with the file name. Once the file has been copied (you provide an InputStream in the REST API call) the method should return. The REST API call to upload the debian file will have a return code of 200 if the request succeeds. The response string associated with the debian package upload is an empty JSON structure.

```java
// communication with the administrative component
@Reference(policy = ReferencePolicy.DYNAMIC,
    cardinality = ReferenceCardinality.OPTIONAL_UNARY)
private volatile AdminRest adminRest;

... File unzipDir = appService.getAppUnzipDir(APP_ID, APP_VERSION);
File deb = new File(unzipDir, APP_DEBIAN_FILE);

... BasicHeader fileHeader =
    new BasicHeader("filename", APP_DEBIAN_FILE);
Header[] headers = {fileHeader};
try (InputStream stream =
    Files.newInputStream(Paths.get(deb.toURI()))) {
    URI uri = adminRest.uri(IpAddress.LOOPBACK_IPv4, UPLOAD_PATH);
    ResponseData resp = adminRest.post(adminRest.login(),
        uri, headers, stream);
    if (resp.status() != Response.Status.OK.getStatusCode())
        throw new IllegalStateException(E_UPLOAD + resp.status());
}
```
Installing a Debian Package

The path to install the Debian file is "/" and requires an action string with the action word "install" and the name of the Debian package. The REST API will return a status of 200, and the string associated with the response is JSON structure with the current status of the controller and Admin services (sdnc and sdna).

```java
uri = adminRest.uri(IpAddress.LOOPBACK_IPv4, "/");
// then we need to install it
ResponseData response = adminRest.post(adminRest.login(),
    uri, installRequestBytes());

if (response.status() != Response.Status.OK.getStatusCode())
    throw new IllegalStateException(E_INSTALL + response.status());
```

```java
private byte[] installRequestBytes() throws UnsupportedEncodingException {
    String install = "{ "action": "install", "name": ",\"" + APP_DEBIAN_FILE + ",\""\}"
    return install.getBytes(UTF8);
}
```

Removing the Debian Package

The application must assume the responsibility for removing the Debian package when the application is uninstalled. An application can determine that it is being uninstalled via an AppEventListener or by checking the current state of the application in a components deactivate method.

An application should consider the differences between “DISABLED” and “UNINSTALLED”. When a user disables an application, the application manager removes that application from the OSGi runtime environment, but the physical files that constitute that application are not removed from disk. If a user decides to “ENABLE” an application that has been disabled, then it is re-introduced into the OSGi runtime environment. When in a “DISABLED” state there is no java code from that application executing (at least the intent is that there is no java code executing). An application should determine if it is proper to remove (or shutdown) the installed Debian package when it is being disabled. When an application is uninstalled, it is the responsibility of the application to remove the installed Debian.
App Event Listener

If an application registers an AppEventListener with the application service then it can receive notification of pending uninstall or disable actions. These notifications are made prior to shutting down the application in the OSGi runtime environment. The AppEventListener registered by the application will hear of any application event for all installed applications. To use this method to determine when an application should remove an installed debian package, it must filter the callback event for just the application id, and then look at the type of event.

```java
private final AppEventListener listener = new AppListener();

@Activate
protected void activate() throws Exception {
    appService.addAppEventListener(listener);
}

private final class AppListener implements AppEventListener {
    @Override
    public void handleAppEvent(ApplicationEventType event, Application app) {
        if (app.id().equals(APP_ID)) {
            if (event.equals(ApplicationEventType.UNINSTALLING) ||
                event.equals(ApplicationEventType.DISABLING)) {
                removeDebian();
            }
        }
    }
}
```

Uploading and Installing the Debian Package

The path to remove the debian file is "/" and requires an action string with the action word “uninstall” and the name of the debian package. The REST API will return a status of 200, and the string associated with the response is JSON structure with the current status of the controller and admin services (sdnc and sdna).

```java
URI uri = adminRest.uri(IpAddress.LOOPBACK_IPv4, "/");
    // then we need to uninstall it
    ResponseData response = adminRest.post(adminRest.login(),
        uri, uninstallRequestBytes());
```
private byte[] uninstallRequestBytes() throws UnsupportedEncodingException {
    String uninstall = "{ "action": "",
    "uninstall", "name": "",
    "" + APP_DEBIAN_FILE + ""};
    return uninstall.getBytes(UTF8);
}
6 Sample Application

The following information describes how to create a complete sample application to show how all the parts fit together, using various parts of the SDN Controller framework.

The SDK provides a tool to generate a skeletal application project structure as a starting template for custom projects. This tool automates the steps described in the following information. Thus, if you prefer (and it is recommended) to use the application generator tool to create an application workspace go directly to Application Generator (Automatic Workspace Creation) on page 144. Note that the application generated by the tool does not provide an actual device monitoring implementation, but merely a skeletal project structure.

This example uses something that is complex enough to show the various services and basic API operations, but not something that gets bogged down with details. It also uses a domain that is familiar to everyone working with SDN so to concentrate on how to work with the HP VAN SDN Controller SDK, not on what the application domain is all about.

Application Description

For this example, we use a domain that is easily understood and that everyone can relate to: An application that monitors reachability status of Open Flow switches. The application provides a simple view to display the current status of the discovered Open Flow switches and it offers a REST API to request discovered devices information. This conceptual domain includes the Open Flow switch which contains information like: IP Address, MAC Address, Friendly Name and Reachability Status (Online, Offline).

Obviously in the real world there would be many more model objects, relations, considerations and much more complexity. This example defines something complex enough to be interesting and touch on the important points, but simple enough to maintain the focus on the HP VAN SDN Controller.

You can get the complete sample application source code from the HP VAN SDN Controller SDK.

Creating Application Development Workspace

The first step to develop an SDN application is creating the application development workspace. The workspace is the set of source projects and configuration files that is compiled, packaged and deployed to the SDN controller.

For the sample application the information from Table 4 is used. In order to create a workspace for a different application it would be necessary to update all appearances of the information shown in Table 4.
Table 4 Sample Application Information

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Name</td>
<td>Health Monitor</td>
</tr>
<tr>
<td>Application Short Name</td>
<td>hm</td>
</tr>
<tr>
<td>Company</td>
<td>Hewlett-Packard</td>
</tr>
<tr>
<td>Company Short Name</td>
<td>hp</td>
</tr>
</tbody>
</table>

The following information describes the how to manually create the application workspace.

Creating Application Directory Structure

Source projects and configuration files will be organized in a directory structure. Any structure works but one similar to the one suggested in Figure 37 is recommended. Table 5 describes the folders under the main application directory (health-monitor for this sample application). Table 5Figure 38 shows the application module dependencies.

Figure 37 Application Directory Structure

Table 5 Application Folders

<table>
<thead>
<tr>
<th>Folder</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hm-app</td>
<td>Configuration Folder</td>
<td>It contains the application deployment OSGi plan, the application descriptor and a POM file used to generate the installable application.</td>
</tr>
<tr>
<td>hm-root</td>
<td>Configuration Folder</td>
<td>Keeps the parent pom.xml file which contains common properties and dependencies to all modules.</td>
</tr>
<tr>
<td>hm-model</td>
<td>Module / Source Code Project</td>
<td>Defines the model objects to use across all application levels. All other projects will depend on this one. This project could be exported (public) if the application will expose services to be consumed by other applications.</td>
</tr>
<tr>
<td>hm-api</td>
<td>Module / Source Code Project</td>
<td>Defines the application’s API or application’s services. This project could also be exported (public) if the application will expose services to be consumed by other applications.</td>
</tr>
</tbody>
</table>
Creating Configuration Files

This section describes the different configuration files that have to be created in order to properly build and package the application so it can be deployed in the HP VAN SDN Controller.

Root POM File

The application root or parent pom.xml file, for which a template can be found in the HP VAN SDN Controller SDK, allows defining common properties across the application’s source projects pom.xml files. It also offers a single entry point to build the entire application. This POM file is auto generated if the application generator tool introduced in Application Generator (Automatic Workspace Creation) is used to generate the application.
Under the application root folder (`hm-root`) create the application parent POM file using the template from the HP VAN SDN Controller SDK. The following list shows the root `pom.xml` after updating the template with Table 4.

Sample Application Root POM File:

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.hp.hm</groupId>
  <artifactId>hm-root</artifactId>
  <packaging>pom</packaging>
  <version>1.0.0-SNAPSHOT</version>
  <name>hm-root</name>
  <description>Health Monitor SDN Application</description>

  <modules>
    <module>../hm-model</module>
    <module>../hm-api</module>

    <module>../hm-bl</module>
    <module>../hm-rs</module>
    <module>../hm-ui</module>
    <module>../hm-app</module>
  </modules>

  <properties>
    <hp-util.version>6.32.0</hp-util.version>
    <sdn.version>2.3.2</sdn.version>
  </properties>

  <!-- Remaining content same as in template -->
  ...
</project>
```

Module POM File

The application module (Source code project) `pom.xml` file, for which a template can be found in the HP VAN SDN Controller SDK, allows creating the Eclipse project and compiling the module. This POM file is auto generated if the application generator tool introduced section Application Generator (Automatic Workspace Creation) is used to generate the application.

Under each application module (or source code project) from Table 5 create the module POM file using the template from the HP VAN SDN Controller SDK. The following list shows the `hm-api pom.xml` after updating the template with Table 4. A `pom.xml` for each application module must be created under the module’s folder.
Sample Application Module POM File:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <parent>
    <groupId>com.hp.hm</groupId>
    <artifactId>hm-root</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <relativePath>../hm-root/pom.xml</relativePath>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>hm-api</artifactId>
  <packaging>bundle</packaging>
  <name>hm-api</name>
  <description>Health Monitor - API Bundle</description>

  <dependencies>
  </dependencies>
</project>
```

Application Descriptor

The application descriptor defines META-data that allows the controller to validate the application being installed.

Under the application app folder (`hm-app`) create the application descriptor using the template from the HP VAN SDN Controller SDK. The following list shows the sample application descriptor (`hm-app/hm.descriptor`) after updating the template with the application information.

Application Descriptor:

```xml
id=com.hp.hm
name=Health-Monitor
version=1.0.0-SNAPSHOT
vendor=Hewlett-Packard
description=Health Monitor SDN Application
```

The Application `id` property may contain alpha-number characters and the period, underscore, and dash characters as long as it is unique across installed applications. The Application `name` property’s value must follow the rules for a properly formatted Java properties file key value; however consider that this value is the one that appears in the SDN controller’s Applications view. The `Version` attribute must be a valid OSGi version number. A valid OSGi number is composed of the following: `major_.minor_.micro_.alpha_numeric_quantifier` (e.g. `6.24.0.build64` or `6.27.0.0`). The Application `order`, `scoped`, `atomic`, `vendor`, and `description` properties are all optional. The `scoped` and `atomic` properties must be true or false. The `vendor` and `description` properties must follow the rules for a properly formatted Java properties file key value, similar to
the name property. The order property must contain a comma separated list of bundle symbolic names indicating the order each bundle should be started in. The first bundle in this list is started by OSGi first, whereas the last bundle in the list is started by OSGi last.

**Application Packaging POM File**

The installable application is a simple .zip file containing the output .jar files generated at the target directory at compile time of each application module (“.../health-monitor/hm-api/target/hm-api-1.0.0.jar” for example).

This application.zip file is automatically generated after building the application if the application generator tool was used to create the application, see Application Generator (Automatic Workspace Creation) on page 144. The application zip file can be found under the ~/hm-app/target.

A POM file can be created to automatically produce the application package or zip file. Under the application app folder (hm-app) create the application packaging pom.xml file using the template from the HP VAN SDN Controller SDK. The following list shows an example for the sample application.

**Sample Application Packaging POM File:**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">
  <parent>
    <groupId>com.hp.hm</groupId>
    <artifactId>hm-root</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <relativePath>../hm-root/pom.xml</relativePath>
  </parent>

  <modelVersion>4.0.0</modelVersion>
  <artifactId>hm-app</artifactId>
  <packaging>pom</packaging>
  <name>hm-app</name>
  <description>Health Monitor - application packaging module</description>

  <dependencies>
    <dependency>
      <groupId>com.hp.hm</groupId>
      <artifactId>hm-model</artifactId>
      <version>${project.version}</version>
    </dependency>
    <dependency>
      <groupId>com.hp.hm</groupId>
      <artifactId>hm-api</artifactId>
      <version>${project.version}</version>
    </dependency>
  </dependencies>
</project>
```
<dependency>
  <groupId>com.hp.hm</groupId>
  <artifactId>hm-bl</artifactId>
  <version>${project.version}</version>
</dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <artifactId>maven-antrun-plugin</artifactId>
      <executions>
        <execution>
          <id>package-app</id>
          <phase>package</phase>
          <configuration>
            <tasks>
              <mkdir dir="target/bundles" />
              <copy todir="target/bundles/" flatten="true">
                <fileset dir="${user.home}/.m2/repository/com/hp/hm/">
                  <include name="hm-model/${project.version}/hm-model-${project.version}.jar"/>
                  <include name="hm-api/${project.version}/hm-api-${project.version}.jar"/>
                  <include name="hm-bl/${project.version}/hm-bl-${project.version}.jar"/>
                  <include name="hm-rs/${project.version}/hm-rs-${project.version}.war"/>
                  <include name="hm-ui/${project.version}/hm-ui-${project.version}.war"/>
                </fileset>
              </copy>
              <fileset dir="${basedir}" includes="hm.descriptor" />
            </tasks>
          </configuration>
          <goals>
            <goal>run</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
</project>

143
Creating Module Directory Structure

At this point there should be a pom.xml file under each folder listed in Table 5. Each application module folder is also a source code project, so a few more subdirectories must be created in order to properly generate the Java Eclipse projects. For each application module create the directory structure as displayed in Figure 39.

**Figure 39 Application Module Directory Structure**

The application development workspace is now completed.

Application Generator (Automatic Workspace Creation)

The HP VAN SDN Controller SDK also contains a utility to generate a skeletal application project structure as a starting template for your custom projects - automatizing all previous steps to create the application workspace. The generated application builds and installs into the HP VAN SDN Controller without any modifications.

The tool allows you to tailor your template application using the parameters listed in Table 6.

**Table 6 Sample Application Generator Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--directory</td>
<td>Target directory where the application code is to be generated</td>
</tr>
<tr>
<td>--app</td>
<td>One word application name in lower-case, for example, 'hm'</td>
</tr>
<tr>
<td>--company</td>
<td>One word company name in lower-case, for example, 'hp'</td>
</tr>
<tr>
<td>--subject</td>
<td>One word short subject in camel-case, for example, 'OpenFlowSwitch'</td>
</tr>
<tr>
<td>--app-name</td>
<td>Optional full application name, for example, 'Health Monitor'</td>
</tr>
</tbody>
</table>
To protect any existing code customizations, the ‘directory’ parameter needs to denote a new
directory, that is, one that does not exist yet.

The ‘app’ and ‘company’ parameters need to be suitable for use in Java package names and
therefore should be all lowercase and not contain any spaces or special characters. Similarly, the
‘subject’ parameter needs to be suitable for use in Java class names and therefore should be in
camel-case and not contain any spaces or special characters either.

The following command shows how to use the application generator tool to build the sample
application.

NOTE:

The target directory must not contain spaces if the application generator is used.

```
$ bin/gen-sdn-app --directory /dev/sdm-apps/health-monitor --template skeleton \
   --app hm --company hp --subject Switch --app-name "Device Health Monitor" \
   --company-name "Hewlett Packard" --description "Application for monitoring health of network devices." --rest-path switches
```

When executing the command with no parameters the command’s documentation is displayed,
which is very useful since the number of parameters is quite big and it is hard to remember them.

The template sample application is ready to build and install. It serves as a good starting point to
new applications development.

To build the application simply change the working directory to the root module and use maven to
build the application as described below.

When Maven is finished, the application zip file can be found under the ~/sdn-hm/hm-app/target directory.  Use the SDN Controller GUI as described in Installing the Application on page 147, to directly upload the application zip file and then ignite it.

Eclipse IDE project files can also be created automatically as described in Creating Eclipse Projects.

### Creating Eclipse Projects

Eclipse projects can be generated by executing the following Maven command from the
application root directory (~/dev/sdn-apps/health-monitor/hm-root for the sample application in Linux):

```
$ mvn eclipse:eclipse
```

Once the maven command completes, the projects can be imported from within the Eclipse IDE, see Importing Java Projects on page 243.
Updating Project Dependencies

The command described in Creating Eclipse Projects on page 145 creates the Eclipse projects resolving all dependencies defined in the POM files. Once the projects have been created and imported into Eclipse, the same command may be used to maintain dependencies.

Execute the command when a dependency is added to the POM file or removed, and then just refresh the projects within Eclipse.

Building the Application

In order to build the application, execute the following command from the application root directory (~/.dev/sdn-apps/health-monitor/hm-root for the sample application in Linux):

$ mvn clean install

Refer to Troubleshooting on page 251 in case of troubles building the application. When the Maven’s build process is completed the application zip file (hm-*.zip) can be found under the target directory of the application’s app module - /health-monitor/hm-app/target. Use the SDN Controller GUI as described in Installing the Application on page 147 to directly upload the application zip file. In order to properly compile source projects they must have at least one Java class.

NOTE

If using the Sample Application Generator to create an application the application modules already contain source files so skip the rest of the section, see Application Generator (Automatic Workspace Creation) on page 144.

If the application workspace was created manually, the application modules are probably empty; thus a class that acts as a seed has to be created on each application module. The class can be as simple as the one shown in the following listing. However, even though the seed classes are temporal and is later replaced by real code, it is convenient using the correct java packages; Table 7 lists suggestions.

**Application Module Seed Java Class:**

```java
package com.hp.hm.api;

/**
 * Place holder to allow the module to be properly compiled and packaged.
 * TODO: Remove this class when real code is added to the module.
 */

public class Seed {
}
```

<table>
<thead>
<tr>
<th>Module</th>
<th>Recommended Package</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7 Suggested Java Packages
Installing the Application

For this example the SDN Controller GUI will be used to deploy the application. It is assumed a test machine was already created, for more information see Test Environment on page 7.

1. Login to the SDN controller as `sdn` user using the following URL “https://SDN_CONTROLLER_ADDRESS:8443/sdn/ui/” as shown in Figure 40. See Authentication Configuration on page 7 to determine the right credentials.

Figure 40 SDN Controller Login Page

2. Click the New tool bar action as illustrated in Figure 41.
3. Upload and install the application as illustrated in Figure 42.
4. Click Browse button to select the application zip file (hm-*.zip in our example) and select Upload; when the upload operation is completed the dialog should load the application’s META-data defined in the application descriptor file created in Application Descriptor on page 141.
5. Click Deploy to install the application.

At this point the application should be part of the applications table and should be installed. To uninstall the application execute the uninstall tool bar action in the same view, as shown in Figure 43.
Application Code

The following information walks through the code and shows how to implement the application. This is useful as it illustrates different services in action.

Space doesn’t permit implementing the entire application, however this shows the major parts, and finishing the implementation is a matter of creating a variation of what is shown. Javadocs will be omitted to save space, however they are important and must be provided in production code. Some code samples will contain comments (in green color) to assist illustrations; these comments are not meant to remain in a real application though. Some lines of code are highlighted (in yellow color) to denote an important difference with previous illustrations of the same code. You can get the complete sample application source code from the HP VAN SDN Controller SDK [18].

NOTE

When the Application Generator (Automatic Workspace Creation) is used to create an application, the application modules already contain source files that follow practices described in the following information. Thus, the generated application can be used as a starting point.

It’s important to note that some parts of the illustrated code are just suggestions (like the way model objects are implemented); you are free to apply any technique and style you prefer; however, the code illustrated follows the same philosophy as the controller’s so it helps to understand the way the controller’s services are structured.
Defining Model Objects

The application requires some standard data structures that act as transfer objects [31]. This example uses a Switch data structure to hold all the information about the Open Flow Switch, shown in the following listing (note that a better name would be OpenFlowSwitch, but a shorter name was selected due to space limitations illustrating code samples).

Switch.java:

```java
package com.hp.hm.model;
import java.util.UUID;
import com.hp.api.Id;
import com.hp.api.Transportable;
import com.hp.sdn.BaseModel;
import com.hp.sdn.auditlog.AuditLogEntry;
import com.hp.sdn.Model;

public class Switch extends Model<Switch> {
    ...

    private String name;

    public Switch() {
        super();
    }

    public Switch(String name) {
        super();
        this.name = name;
    }

    public Switch(Id<Switch> id, String name) {
        super(id);
        this.name = name;
    }

    // Implement setters and getters for mutable fields: name

    // Good practice to override the following methods on transport objects:
    // equals(Object), hashCode() and toString()
    ...
}
```

The Switch model object implements the Transportable interface which is part of the HP VAN SDN Controller Framework; Model, which extends AbstractModel, offers a partial implementation of this interface. In order to use such an interface and its partial implementation, the root POM file must resolve the dependencies. The root POM file is used to resolve the dependencies because these
required modules are used at all application levels (Presentation logic, controller logic, business logic, cross-cutting logic, and so on), thus all the application modules will depend on them. Later it’ll be shown how specific dependencies to certain modules are added into the specific module POM file.
NOTE
At this point AbstractModel (which Model extends) and Transportable are used just to denote that Switch follows the data transfer object pattern [31]. AbstractModel is a convenient partial implementation because it properly overrides equals() and hashCode() methods. However, if the HP VAN SDN Controller’s persistence framework is used to persist data, then data transfer objects take an explicit role and they must follow certain hierarchical constraints. At the time this document was written, the HP VAN SDN Controller made use of two different persistence frameworks: Relational (JPA) and Non-Relational (Cassandra). These frameworks will be unified in the future, but at this phase two different interfaces for transfer objects are defined: one to use in relational models and one to use in non-relational. See Introduction.

In a network managed by a controller, the controller itself stands out to be a single point of failure. Controller failures can disrupt the entire network functionality. HP VAN SDN Controller Distributed Coordination infrastructure provides various mechanisms that controller applications can make use of in achieving active-active, active-standby Distributed Coordination paradigms and internode communication. The Distributed Coordination infrastructure provides 2 services for the applications to develop Distributed Coordination aware controller modules.

- Controller Teaming
- Distributed Coordination Service

Following figure describes the communication between the controller applications and the HP VAN SDN Controller Distributed Coordination sub-systems. “App1 – 1” indicates instance of application 1 on controller instance 1. Distributed services, ensures the data synchronization across the controller cluster nodes.

Figure 44 Application view of Coordination Services
Open the `hm-root/pom.xml` file and add the XML extract from the following list to the `<dependencies>` node. After updating the POM file update the Eclipse project dependencies (see Updating Project Dependencies on page 146).

**HP SDN Controller Framework Common Dependencies:**

```xml
<dependency>
  <groupId>com.hp.util</groupId>
  <artifactId>hp-util-misc</artifactId>
  <version>${hp-util.version}</version>
</dependency>

<dependency>
  <groupId>com.hp.util</groupId>
  <artifactId>hp-util-api</artifactId>
  <version>${hp-util.version}</version>
</dependency>

<dependency>
  <groupId>com.hp.util</groupId>
  <artifactId>hp-util-ip</artifactId>
  <version>${hp-util.version}</version>
</dependency>

<dependency>
  <groupId>com.hp.sdn</groupId>
  <artifactId>sdn-common-model</artifactId>
  <version>${sdn.version}</version>
</dependency>
```

If the application offers, and this sample application does, a way to retrieve model objects—in this example Open Flow switches—based on some kind of filter, then it is a good practice to create a POJO class [19] that represents the filter. Creating such a class will help decoupling the service consumer from the way filtering is implemented in lower level layers (like the data store service or database). The HP VAN SDN Controller Framework provides a set of classes that represent filter conditions which can be used to compose a filter.

These classes include:

- **Comparable Condition**—Used to represent the following conditions: Less than, less than or equal to, equal, greater than or equal to and greater than.
- **Equality Condition**—Used to represent the following conditions: Equal and unequal.
- **Interval Condition**—Used to represent the following conditions: In and not in.
- **Set Condition**—Used to represent the following conditions: In and not in.
- **String Condition**—Used to represent the following conditions: Equal, unequal, starts with, contains and ends with.

Based on these conditions we will create a filter for the Open Flow switch class as illustrated in the following listing.

**SwitchFilter.java:**

```java
package com.hp.hm.model;
```
import com.hp.util.filter.EqualityCondition;
import com.hp.util.filter.StringCondition;
...
public class SwitchFilter {

    private StringCondition nameCondition;
    ...
    // Implement setters and getters for all conditions.
    // Good practice to override toString()
}

The following listing depicts a usage example: Create a filter to retrieve all open flow switches with a name that contains the text 'My Switch.'

Switch Filter Usage Example:
    SwitchFilter filter = new SwitchFilter();
    filter.setNameCondition(new StringCondition("My Switch",
        StringCondition.Mode.CONTAINS));

Following a similar approach, create a Java enumeration to represent the sort possibilities in which open flow switches can be retrieved. This helps decoupling the service consumer from the way sorting is implemented in lower level layers (like column names in a database). The following listing shows the Open Flow Switch sort possibilities and the next listing depicts a usage example: When retrieving switches the primary order shall be the Name ascending.

SwitchSortKey.java:
    package com.hp.hm.model;

    public enum SwitchSortKey {
        NAME
    }

Switch Sort Specification Usage Example:
    SortSpecification<SwitchSortKey> sort =
        new SortSpecification<SwitchSortKey>();
    sort.addSortComponent(SwitchSortKey. NAME, SortOrder.ASCENDING);

Model Objects Unit Test

The HP VAN SDN Controller Framework offers some utilities to facilitate writing unit tests.

Even though The Data Transfer Object [31] is a pattern while a JavaBean [32] is a specification, consider a Data Transfer Object as a java bean which is transported across tiers. The Data Transfer Object pattern is used as a light-weight method of transferring data between layers. Thus, use the Bean Test utilities provided by the HP VAN SDN Controller Framework to test the transfer objects. The following listing illustrates the utility classes provided by the HP VAN SDN Controller Framework that can be used to test model objects. For the complete test code see the sample
application source code included with the HP VAN SDN Controller SDK. SwitchTest should be located under `hm-model/src/test/java/com/hp/hm/model` directory.

SwitchTest.java:
```java
package com.hp.hm.model;

import com.hp.test.BeanTest;
import com.hp.test.EqualityTester;
import com.hp.test.SerializabilityTester;

public class SwitchTest {
    ...
    @Test
    public void testGettersAndSetters() throws Exception {
        Switch device = //... create an instance
        BeanTest.testGettersAndSetters(device);
    }

    @Test
    public void testEqualsAndHashcode() {
        Switch base = //... create the base object
        Switch equals1 = //... create an object equal to the base
        Switch equals2 = //... create an object equal to the base
        Switch unequal = //... create an object unequal to the base

        EqualityTester.testEqualsAndHashCode(base, equals1,
                                            equals2, unequal);
    }

    @Test
    public void testSerialization() {
        Switch device = //... create with attributes set to non-null values
        SerializabilityTester.testSerialization(device);
    }
}
```

**BeanTest** utility class—A rudimentary facility for generic testing of basic bean getter and setter functionality. It uses reflection to locate matching getter/setter pairs in the supplied bean instance.

**EqualityTester** class—Verifies the equivalence relation on non-null object references as documented in the `Java Object.equals(Object)` method; it follows the `equals` contract and makes sure its properties hold: reflexive, symmetric, transitive, consistent and non-null reference.

**SerializabilityTester** class—Serializes and deserializes the object being tested looking for serialization failures (`java.io.NotSerializableException`) which are thrown when an instance is required to have a Serializable interface. It is crucial to set a non-null value to all non-transient attributes in the object under test, otherwise serialization failures won’t be detected.
Creating Domain Service (Business Logic)

The following information defines a service to provide Open Flow Switches functionality (The sample application’s business logic). This service basically provides operations to create, read, update and delete open flow switches (CRUD operations).

Service API

Service API abstracts the business logic implementation by defining an API that clients or consumers use in order to interact with Open Flow switches. This API will act as the Open Flow Switch service contract. The following listing shows the Open Flow Switch service API which should be created under hm-api module.

SwitchService.java (Sample Application Service API):

```java
package com.hp.hm.api;
import java.util.Collection;
import java.util.UUID;
import com.hp.api.Id;
import com.hp.api.NotFoundException;
import com.hp.hm.model.Switch;
...
public interface SwitchService {
    public Switch create(String name);
    public Collection<Switch> getAll();
    public Switch get(Id<Switch, UUID> id);
    public void delete(Id<Switch, UUID> id);
}
```

Services expose methods that use transfer objects, primitive types, object value types and common data structures in their signatures; thus, these entities become part of the API and they remain the same no matter the implementation we choose for our services.

The Switch service depends on the hm-model module because model objects are defined there, thus the hm-api POM file needs to resolve the dependencies. Open the hm-api/pom.xml file and add the XML extract from the following listing to the <dependencies> node. After updating the POM file update the Eclipse project dependencies (see Updating Project Dependencies on page 146).

Application Model Dependency:

```xml
<dependency>
    <groupId>com.hp.hm</groupId>
    <artifactId>hm-model</artifactId>
    <version>${project.version}</version>
</dependency>
```
Service Implementation

Implementation of our API or services will be located at the `hm-bl` module. As with `hm-api`, the business logic module will also depend on the `hm-model`, as well as on the `hm-api` module. So open the `hm-bl/pom.xml` file and add the XML extract from the following listing to the `<dependencies>` node; after updating the POM file update the Eclipse project dependencies (see Updating Project Dependencies on page 146).

Application API Dependency:

```xml
<dependency>
    <groupId>com.hp.hm</groupId>
    <artifactId>hm-model</artifactId>
    <version>${project.version}</version>
</dependency>
<dependency>
    <groupId>com.hp.hm</groupId>
    <artifactId>hm-api</artifactId>
    <version>${project.version}</version>
</dependency>
```

Now, create the Open Flow Service implementation, name it `SwitchManager` – The suffix `Manager` is used to denote services implementations. The following listing shows an extract of the implementation. For the moment it returns fake data, in later information the fake data is replaced by more realistic data.
package com.hp.hm.impl;
...

public class SwitchManager implements SwitchService {

    @Override
    public Switch create(String name) {
        Switch s = new Switch(name);
        if (isEmpty(s.name())) {
            s.setname("Switch-" + s.getId().getValue().toString());
        }
        store.put(s.getId(), s);
        return s;
    }

    @Override
    public Collection<Switch> getAll() {
        synchronized (store) {
            return Collections.unmodifiableCollection(store.values());
        }
    }

    @Override
    public Switch get(Id<Switch, UUID> id) {
        synchronized (store) {
            Switch s = store.get(id);
            if (s == null)
                throw new NotFoundException("Switch with id " + id + " not found");
            return s;
        }
    }

    @Override
    public void delete(Id<Switch, UUID> id) {
        synchronized (store) {
            Switch s = store.remove(id);
            if (s == null)
                throw new NotFoundException("Switch with id " + id + " not found");
        }
    }
}
Providing Services with OSGi Declarative Services

The OSGi standard component framework, called Declarative Services [33], is used to create component-oriented applications. It is called declarative because there is no need to write explicit code to publish or consume services.

A component describes functional building blocks that are typically more coarse-grained than what we normally associate with objects.

These building blocks are typically business logic; they provide functionality via interfaces. Conversely, components may consume functionality provided by other components via their interfaces. A component framework is used to execute components.

A component model describes what a component looks like, how it interacts with other components, and what capabilities it has (such as lifecycle or configuration management). A component framework implements the runtime needed to support a component model and execute the components.

The general approach for creating an application from components is to compose it. This means you grab the components implementing the functionality you need and compose them (match required interfaces to provided interfaces) to form an application. Component compositions can be declarative, such as using some sort of composition language to describe the components and bindings among them.

By using components applications can be created easily and quickly by snapping them together from readily available, reusable components. Components promote separation of concerns and encapsulation with its interface based approach. This enhances the reusability of your code because it limits dependencies on implementation details. Another worthwhile aspect of an interface-based approach is substitutability of providers. Because component interaction occurs through well-defined interfaces, the semantics of these interfaces must themselves be well defined. As such, it’s possible to create different implementations and easily substitute one provider with another.

The type of component model defined by OSGi is called service-oriented component model which rely on execution-time binding of provided services to required services using the service-oriented interaction pattern [33].

Continuing with the example, the SwitchService from the Service API on page 156 will be published via SwitchManager from the Service Implementation on page 157 so it is available to be consumed by other components.

SwitchManager is a Java object not bound by any restriction other than the service interface it implements and those forced by the Java Language Specification; similar to a POJO [19]. Since OSGi declarative services require a component to be annotated and to implement some methods to bind/unbind other dependency components, a proxy component will be introduced (that follows the proxy pattern [34]) to deal with OSGi allowing the business logic to be separated from the OSGi restrictions. The following listing shows the OSGi service component used to publish SwitchService via OSGi declarative services. The implementation of the OSGi component should also be located in the hm-bl module.
NOTE

The usage of `SwitchComponent` may be omitted and directly annotate `SwitchManager` if preferred. The generated example application does not provide a `SwitchComponent.java` file.

SwitchComponent.java (Sample Application OSGi Service Component):

```java
package com.hp.hm.impl;

import org.apache.felix.scr.annotations.Component;
import org.apache.felix.scr.annotations.Service;
...
@Component
@Service
public class SwitchComponent implements SwitchService {

    private SwitchService delegate;

    public SwitchComponent() {
        delegate = new SwitchManager();
    }

    @Override
    public Switch create(String name) {
        return delegate.add(name);
    }

    @Override
    public Collection<Switch> getAll() {
        delegate.getAll();
    }

    @Override
    public Switch get(Id<Switch, UUID> id) {
        return delegate.get(id);
    }

    @Override
    public void delete(Id<Switch, Long> id) {
        delegate.delete(id);
    }
}
```

`SwitchComponent` is annotated with `@Component` to make it part of the OSGi component management framework (lifecycle management) and thus it is allowed to consume other
components. It is also annotated with @Service to denote this component should be published so it is consumed by other components.

As mentioned above, don’t write explicit code to publish or consume services. Thus, SwitchService is ready to be published when the application is installed into the HP VAN SDN Controller.

Verifying Published Services Using Virgo Admin Console

In order to verify our service is actually published we may use the Virgo Admin Console (This console may be also used to uninstall applications). First build and install the application as described in Building the Application and Installing the Application sections.

NOTE

The following information describes the process using two different versions of Virgo [8] container. The latest HP VAN SDN Controller was upgraded to use the newer version; however this information describes the way of verifying published services using an older version because unfortunately Virgo dropped the “Published Services” information in the new version and now it is not possible to see published services unless they are already consumed. Thus the old version is illustrated as a good reference.

Virgo 3.5.0

Open a browser at https://[SDN_CONTROLLER_ADDRESS]:8443/admin/web/info/overview.htm and follow the steps described by Figure 46, Figure 47, Figure 48 and Figure 49. Use ‘admin’ as user and ‘sdn’ as password. When everything works as expected the SwitchService entry under ‘Published Services’ is seen as illustrated in Figure 49.

Figure 46 Virgo 3.5.0 Admin Console
Figure 47  Virgo 3.5.0 Admin Console Artifacts

Figure 48  Virgo 3.5.0 Admin Console Application Plan
Since Virgo dropped the “Published Services” section illustrated in Figure 49, it is not possible to see published services unless they are already consumed. At this point in this example, the service is not being consumed so it is not possible to see it as published service. However, this section illustrates the way of verifying consumed services when the SwitchService is already being consumed by the hm-rs module.

Open a browser at https://[SDN_CONTROLLER_ADDRESS]:8443/admin and follow the steps described by Figure 50, Figure 51, Figure 52, and Figure 53. Use ‘admin’ as user and ‘sdn’ as password. If everything worked as expected you should be able to see the SwitchService entry under ‘Published Services’ section as illustrated in Figure 53.
Figure 50 Virgo 3.6.1 Admin Console

Figure 51 Virgo 3.6.1 Admin Console Artifacts
Figure 52 Virgo 3.6.1 Admin Console Application Plan

Figure 53 Virgo 3.6.1 Admin Console Business Logic Bundle Relationships by Service
Consuming Services with OSGi Declarative Services

OSGi Declarative Services may also be used to consume other services: injecting references of other components (Dependency components) into our components (via OSGi’s dependency-injection framework).

Assume the business service implementation (SwitchManager) depends on the SystemInformationService - a service provided by the HP VAN SDN Controller to request system information such as the system IP Address, and so on. Also assume the relation is mandatory meaning the service cannot operate without such dependency and thus it should not be published until the dependency is satisfied (SystemInformationService is available and has been injected into SwitchManager).

Assume SwitchManager depends on the AlertService - a service provided by the HP VAN SDN Controller to post alerts. However, assume this dependency is optional, which means SwitchManager is activated and published even though the AlertService is not.

Since SwitchManager is not tied to OSGi, adding mandatory dependencies is as simple as defining constraints at construction time. Mutators are used to set optional dependencies (a better way to handle optional-dependencies is to use the decorator pattern [34] to decorate business logic with optional services). The following listing shows the modified SwitchManager which now depends on SystemInformationService and AlertService. Dependency services are defined in a different module thus the business logic module needs to declare such dependencies in its POM file. Open the hm-bl/pom.xml file and add the XML extract from SystemInformationService listing to the <dependencies> node; after updating the POM file update the Eclipse project dependencies (see Updating Project Dependencies on page 146).

Dependent SwitchManager.java:

```java
package com.hp.hm.impl;

import com.hp.sdn.adm.alert.AlertService;
import com.hp.sdn.adm.system.SystemInformationService;
...
public class SwitchManager implements SwitchService {

  // Mandatory dependency.
  private final SystemInformationService systemInformationService;

  // Optional dependency. NOTE: A better design would use the decorator
  // pattern to decorate business logic with optional services.
  private AlertService alertService;

  public SwitchManager(SystemInformationService systemInformationService) {
    // Mandatory dependencies are set at construction time.
    if (systemInformationService == null) {
      throw new NullPointerException(...);
    }
    this.systemInformationService = systemInformationService;
  }
```

166
public void setAlertService(AlertService alertService) {
    // Mutators are used for optional dependencies.
    this.alertService = alertService;
}

SystemInformationService Module Dependency:
<dependency>
    <groupId>com.hp.sdn</groupId>
    <artifactId>sdn-adm-api</artifactId>
    <version>${sdn.version}</version>
</dependency>

As previously mentioned, SwitchManager is not tied to OSGi so it expects a non-null instance of SystemInformationService (it doesn’t care how the instance is obtained) and code does not need to be included to handle the case when the implementation of SystemInformationService is no longer available. SwitchManager focuses on implementing the business logic. Note how SystemInformationService is an interface, so its implementation may be changed without affecting the business logic.

SwitchComponent will be updated to obtain a reference of SystemInformationService and AlertService via OSGi declarative services. SwitchComponent must deal with the fact that components may come and go, thus the injected references need to be bound and unbound. The following listing shows SwitchComponent consuming the services.

Dependent SwitchComponent.java:

```java
package com.hp.hm.impl;

import org.apache.felix.scr.annotations.Reference;
import org.apache.felix.scr.annotations.ReferenceCardinality;
import org.apache.felix.scr.annotations.Service;
...
@Component
@Service
public class SwitchComponent implements SwitchService {

    @Reference(policy = ReferencePolicy.DYNAMIC,
            cardinality = ReferenceCardinality.MANDATORY_UNARY)
    private volatile SystemInformationService systemInformationService;

    @Reference(policy = ReferencePolicy.DYNAMIC,
            cardinality = ReferenceCardinality.OPTIONAL_UNARY)
    private volatile AlertService alertService;

    // Note: A better design would use the decorator pattern to decorate
    // business logic with optional services. That would allow us to use
    // SwitchService instead of SwitchManager as the delegate type.
```
private SwitchManager delegate;

@Activate
public void activate() {
    // activate() is called after all mandatory dependencies
    // are satisfied
    delegate = new SwitchManager(systemInformationService);
    delegate.setAlertService(alertService);
}

@Deactivate
public void deactivate() {
    delegate = null;
}

protected void bindAlertService(AlertService service) {
    alertService = service;
    // TODO: Decorate the business logic with the optional service.
    if (delegate != null) {
        delegate.setAlertService(service);
    }
}

protected void unbindAlertService(AlertService service) {
    if (alertService == service) {
        alertService = null;
        if (delegate != null) {
            delegate.setAlertService(null);
        }
    }
}

@Override
public Switch add(Switch device) {
    return delegate.add(device);
}

...  
// Follow the same pattern than “add(Switch)” for the
// remaining overridden methods.
}

Dependency services are annotated with @Reference to denote to OSGi to inject a reference into the component. The OSGi’s dependency-injection framework calls bindAlertService(AlertService) method when the service is available (activated) and unbindAlertService(AlertService) when the component providing the implementation of AlertService is deactivated. If no bind/unbind methods are provided (Like in the case of SystemInformationService) OSGi still injects a reference directly into the variable annotated with @Reference. Defining methods to bind/unbind services
allows us to do any pre/post processing when the binding/unbinding takes place - useful when using optional services.

The name for the methods to bind/unbind follows a standard defined by OSGi [5]. The name is composed by the prefix bind/unbind plus the name of the variable in camel case format. Since the variable is called `alertService`, the method to bind must be called `bindAlertService` ("bind" plus the name of the variable with the first letter upper case). The annotation `@Reference` offers an attribute "name" that allows changing the suffix for the bind/unbind methods. Check the OSGi [5] [33] reference for more details.

In order to verify the service is actually consuming `SystemInformationService` use the Virgo Admin Console again as described in Verifying Published Services Using Virgo Admin Console on page 161. When everything works as expected `SwitchService` can be seen as a published Service. If `SwitchService` is published it means it is consuming `SystemInformationService` because a mandatory relation was specified; if `SystemInformationService` was not available then `SwitchService` wouldn’t be published.

Creating a REST API

In the following information RESTful Web Services (or REST API) is created to expose to the outside world functionality provided by the sample application.

REST follows a client-server architecture to achieve separation of concern between the client and the server. The client is not concerned about the internal representation and state diagram of the server, and the server is not concerned about the client logics and states. Instead, the client and the server communicate via a simple uniform interface that is devoid of state information (Stateless) [1].

For HTTP, the client is typically a web browser, but can also be a variety of other software, such as Curl [17], a mobile app, or a desktop app. The server is typically a web container such as a Java Servlet container (Our case), IIS, or Python WSGI container.

The communication between the client and the server must be stateless. That is, a request from a client should not depend on a previous request, as the server does not store client state information. This implies that each client request must contain all the information the server needs to process it.

Creating Domain Service Resource (REST Interface of Business Logic Service)

Table 9 describes the REST API implemented to expose the `SwitchService` functionality from the sample application.

### Table 9 Switch REST API

<table>
<thead>
<tr>
<th>Request</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET /sdn/hm/v1.0/switches/</td>
<td>Lists all switches managed by the application.</td>
</tr>
<tr>
<td>GET /sdn/hm/v1.0/switches/{id}</td>
<td>Gets the switch with the given id.</td>
</tr>
<tr>
<td>POST /sdn/hm/v1.0/switches/</td>
<td>Adds a switch. The request’s data must contain the switch data in JSON format.</td>
</tr>
<tr>
<td>DELETE /sdn/hm/v1.0/switches/{id}</td>
<td>Deletes the switch with the given identity.</td>
</tr>
</tbody>
</table>
Implementation of the REST API is located in the *hm*-rs module. Now, create the Switch REST API which is named *SwitchResource* – The suffix *Resource* is used to denote REST web services. The following listing shows an extract of the resource. For the moment use fake data, in later information replace the fake implementations by more realistic ones. In order to implement REST web services the module needs to declare some dependencies. Open the *hm*-rs/*pom.xml* file and add the XML extract from the REST Module Dependencies listing to the `<dependencies>` node; after updating the POM file update the Eclipse project dependencies (see Updating Project Dependencies on page 146).

SwitchResource.java (REST API):

```java
package com.hp.hm.rs;

import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import com.hp.sdn.rs.misc.ControllerResource;
...
@Path("switches")
public class SwitchResource extends ControllerResource {  

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response getAll() {
        return ok("{"switches":[]}").build();
    }

    @GET
    @Path("{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response get(@PathParam("id") long id) {
        return ok("{"switch":{}}").build();
    }

    @POST
    @Produces(MediaType.APPLICATION_JSON)
    public Response add(String request) {
        return ok("{"switch":{}}").build();
    }

    @DELETE
    @Path("{id}")
    @Produces(MediaType.APPLICATION_JSON)
```
public Response delete(@PathParam("id") long id) {
    return Response.ok().build();
}

**REST Module Dependencies:**

```
<dependency>
    <groupId>com.hp.hm</groupId>
    <artifactId>hm-model</artifactId>
    <version>${project.version}</version>
</dependency>
<dependency>
    <groupId>com.hp.hm</groupId>
    <artifactId>hm-api</artifactId>
    <version>${project.version}</version>
</dependency>
<dependency>
    <groupId>com.hp.util</groupId>
    <artifactId>hp-util-rs</artifactId>
    <version>${hp-util.version}</version>
</dependency>
<dependency>
    <groupId>com.hp.util</groupId>
    <artifactId>hp-util-rs</artifactId>
    <version>${hp-util.version}</version>
    <classifier>tests</classifier>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>com.hp.sdn</groupId>
    <artifactId>sdn-adm-rs-misc</artifactId>
    <version>${sdn.version}</version>
</dependency>
<dependency>
    <groupId>com.hp.sdn</groupId>
    <artifactId>sdn-adm-rs-misc</artifactId>
    <version>${sdn.version}</version>
    <classifier>tests</classifier>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-server</artifactId>
    <version>1.17</version>
    <scope>compile</scope>
</dependency>
```
The hm-rs module needs to be modified so it produces a web application archive (.war file) as output so it is deployed as a web application that serves the RESTful web services for this sample application. Create the file hm-rs/src/main/webapp/WEB-INF/web.xml with the content shown in REST Module Web Application (web.xml) listing. The REST Module Web Application (web.xml) listing configures the Jersey Servlet that handles HTTP requests and dispatches to the right REST API based on the @Path annotations. The highlighted text in the next listing emphasizes the way the application's RESTful web services are registered within the Jersey Servlet.

REST Module Web Application (web.xml):

```xml
<web-app version="1.0" encoding="UTF-8">  
  <web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee" 
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" 
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee 
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">  
    <display-name>Health Monitor REST API</display-name>  
    <servlet>  
      <servlet-name>REST Services</servlet-name>  
      <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>  
      <init-param>  
        <param-name>com.sun.jersey.spi.container.ContainerResponseFilters</param-name>  
        <param-value>com.hp.sdn.rs.misc.CrossDomainFilter</param-value>  
      </init-param>  
      <init-param>  
        <param-name>com.hp.sdn.rs.AllowsDomains</param-name>  
        <param-value>*</param-value>  
      </init-param>  
      <init-param>  
        <param-name>com.sun.jersey.spi.container.ContainerRequestFilters</param-name>  
        <param-value>com.hp.util.rs.auth.AuthJerseyFilter</param-value>  
      </init-param>  
    </servlet>  
  </web-app>  
</web-app>
```
<param-name>exclude-paths</param-name>
<param-value>^\{NONE\}[/\]*(.*)\$</param-value>
</init-param>

<init-param>
  <param-name>
    com.sun.jersey.config.property.resourceConfigClass
  </param-name>
  <param-value>
    com.sun.jersey.api.core.ClassNamesResourceConfig
  </param-value>
</init-param>

<init-param>
  <param-name>
    com.sun.jersey.config.property.classnames
  </param-name>
  <param-value>
    <!— Application REST API -->
    com.hp.hm.rs.SwitchResource

    <!— Application Error Handlers -->

    <!-- Provided Error Handlers -->
    com.hp.sdn.rs.misc.DuplicateIdErrorHandler
    com.hp.sdn.rs.misc.NotFoundErrorHandler
    com.hp.sdn.rs.misc.ServiceNotFoundErrorHandler
    com.hp.sdn.rs.misc.IllegalDataHandler
    com.hp.sdn.rs.misc.IllegalStateHandler
    com.hp.sdn.rs.misc.AuthenticationHandler
  </param-value>
</init-param>

<load-on-startup>0</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>REST Services</servlet-name>
  <url-pattern>/*</url-pattern>
</servlet-mapping>

<filter>
  <filter-name>Token Authentication Filter</filter-name>
  <filter-class>com.hp.sdn.rs.misc.TokenAuthFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>Token Authentication Filter</filter-name>
</filter-mapping>
Next, update the hm-rs module POM file \texttt{hm-rs/pom.xml} with the extract shown in the following listing to generate the .war file during the build process.

\texttt{hm-rs/pom.xml} to generate .war:

\begin{verbatim}
... 
<modelVersion>4.0.0</modelVersion> 
<artifactId>hm-rs</artifactId> 
<packaging>war</packaging> 
... 
<properties> 
  <banned.rs.paths>com.hp.hm.rs</banned.rs.paths> 
  <webapp.context>sdn/hm/v1.0</webapp.context> 
  <web.context.path>sdn/hm/v1.0</web.context.path> 
</properties> 
... 
<build> 
  <plugins> 
    <plugin> 
      <groupId>org.apache.felix</groupId> 
      <artifactId>maven-bundle-plugin</artifactId> 
      <version>2.3.6</version> 
      <extensions>true</extensions> 
      <executions> 
        <execution> 
          <id>bundle-manifest</id> 
          <phase>process-classes</phase> 
          <goals> 
            <goal>manifest</goal> 
          </goals> 
        </execution> 
      </executions> 
      <configuration> 
        <manifestLocation>${project.build.directory}/META-INF</manifestLocation> 
      </configuration> 
    </plugin> 
  </plugins> 
  <supportedProjectTypes> 
    <supportedProjectType>bundle</supportedProjectType> 
    <supportedProjectType>war</supportedProjectType> 
  </supportedProjectTypes> 
  <instructions> 
    <Import-Package> 
      com.sun.jersey.api.core, 
      com.sun.jersey.spi.container.servlet, 
    </Import-Package> 
  </instructions> 
</build> 
\end{verbatim}
If you created an application deployment plan (hm-app/hm.plan), update it (created in Application Deployment Plan on page 110) to deploy the hm-rs module as highlighted in the following listing.

Sample Application Deployment Plan Considering REST Module:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<plan name="health-monitor.plan" version="1.0.0" scoped="false" atomic="false"
    xmlns="http://www.eclipse.org/virgo/schema/plan"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=""
```
http://www.eclipse.org/virgo/schema/plan
http://www.eclipse.org/virgo/schema/plan/eclipse-virgo-plan.xsd

<artifact type="bundle" name="com.hp.hm.hm-model" version="1.0.0.SNAPSHOT "/>
<artifact type="bundle" name="com.hp.hm.hm-api" version="1.0.0.SNAPSHOT "/>
<artifact type="bundle" name="com.hp.hm.hm-bl" version="1.0.0.SNAPSHOT "/>
<artifact type="bundle" name="com.hp.hm.hm-rs" version="1.0.0.SNAPSHOT "/>

Finally update the application packaging POM file hm-app/pom.xml (created in Application Packaging POM File on page 142) with the extract shown in the following listing to include the .war file into the application package.

Sample Application Packaging POM File Including REST Module:

...  
  <build>  
    <plugins>
    <plugin>  
      <artifactId>maven-antrun-plugin</artifactId>
      <executions>
        <execution>
          <id>package-app</id>
          <phase>package</phase>
          <configuration>
            <tasks>
              <mkdir dir="target/bundles" />
              <copy todir="target/bundles/" flatten="true">
                <fileset
dir="${user.home}/.m2/repository/com/hp/hm/">
                  <include name="hm-model/${project.version}/hm-model-${project.version}.jar"/>
                  <include name="hm-rs/${project.version}/hm-rs-${project.version}.war"/>
                </fileset>
               <fileset dir="${basedir}" includes="hm.plan"/>
               <zip destfile="target/hm-${project.version}.zip" basedir="target/bundles"/>
            </tasks>
          </configuration>
        </execution>
      </executions>
    </plugin>
    </plugins>
  </build>
Trying the REST API with curl

The following information illustrates a method to try the REST API created previously in Creating Domain Service Resource (REST Interface of Business Logic Service) on page 169 using curl [17]. See Figure 6 for installation instructions. Build and install the application as described in Building the Application on page 146 and Installing the Application on page 147.

Execute the following command **CURL Authentication Command** to authenticate (And get an authentication token). Then use the authentication token in the **CURL GET Command** to execute a GET on the REST API described in Table 9. Figure 54 shows an execution example using 15.255.126.49 as the SDN controller address. The response returned by SwitchResource can be seen as the output of the **CURL GET Command**.

---

**NOTE**

Use the correct password if it was changed following instructions from Authentication Configuration on page 7.

**CURL Authentication Command:**

```
$ curl --noproxy [SDN_CONTROLLER_ADDRESS] -X POST --fail -ksfL \  
   --url "https://[SDN_CONTROLLER_ADDRESS]:8443/sdn/v2.0/auth" \  
   -H "Content-Type: application/json" \  
   --data-binary "\  
   '{"login":{"user":"sdn","password":"skyline","domain":"sdn"}}'"
```

**CURL GET Command:**

```
$ curl --noproxy [SDN_CONTROLLER_ADDRESS] \  
   --header "X-Auth-Token:[AUTHENTICATION_TOKEN]" \  
   --fail -ksS -L -f \  
   --request GET \  
   --url "https://[SDN_CONTROLLER_ADDRESS]:8443/sdn/hm/v1.0/switches"
```
**RESTful Web Services Unit Test**

Even though at this point the implementation uses fake data, a unit test is shown to illustrate the utility classes provided by the HP VAN SDN Controller SDK; creating good test cases is application dependent and it is out of the scope of this document.

The following listing shows the unit test for `SwitchResource` using the infrastructure class `ClientResourceTest` provided by the HP VAN SDN Controller SDK. `SwitchResourceTest` should be located under `hm-rs/src/test/java/com/hp/hm/rs` directory. New dependencies needed at runtime must be declared in order to properly run the resource test. Open the `hm-rs/pom.xml` file and add the XML extract from the Resource Test Dependencies listing to the `<dependencies>` node; after updating the POM file update the Eclipse project dependencies (see Updating Project Dependencies on page 146).

`SwitchResourceTest.java`:

```java
class SwitchResourceTest extends ControllerResourceTest {

    private static final String BASE_PATH = "switches";

    public SwitchResourceTest() {
        super("com.hp.hm.rs");
    }

    @Override
    @Before
    public void setUp() throws Exception {
        super.setUp();
        // If a specific test case expects a different format, such
        // format will have to be set calling this method.
    }
}
```
ResourceTest.setDefaultMediaTypes(MediaType.APPLICATION_JSON);
}

// When using the inherited methods get(...), post(...), put(...) and
// delete(..) if exceptions are thrown by the Resource (REST) or if the
// returned code is different than 200 (OK) the test fail.

@Test
public void testList() {
    String response = get(BASE_PATH);
    String expectedResponse = path: "{"switches":[]}";
    assureResponseContains(response, expectedResponse);
}

@Test
public void testGet() {
    long idMock = 1;
    String path = BASE_PATH + "/" + idMock;
    String response = get(path);
    String expectedResponse = path: "{"switch":[]}";
    assureResponseContains(response, expectedResponse);
}

@Test
public void testAdd() {
    String jsonRequest = path: "{"switch":[]}";
    String response = post(BASE_PATH, jsonRequest);
    String expectedResponse = path: "{"switch":[]}";
    assureResponseContains(response, expectedResponse);
}

@Test
public void testDelete() {
    long idMock = 1;
    String path = BASE_PATH + "/" + idMock;
    String response = delete(path);
    Assert.assertTrue(response.isEmpty());
}

Resource Test Dependencies:
<dependency>
    <groupId>com.hp.sdn</groupId>
    <artifactId>sdn-common-misc</artifactId>
    <version>${sdn.version}</version>
</dependency>
Domain Service - REST API Integration

Figure 3 illustrates a common pattern used when working with Servlets: The Model-View-Controller (MVC) pattern. In this pattern the Servlet acts as the Controller. As mentioned before, when using RESTful Web Services we don’t directly write Servlets, however a REST API acts as the Controller as well. A normal behavior of a REST API includes:

1. Decode the request—JSON [36] format in our sample application.
2. Call domain services—(Business Logic) to service the request.
3. Encode the result to include in the response—JSON [36] format in our sample application.

This section describes how to integrate the domain service (Business Logic) and the REST API (RESTful web services). The objective is to have the REST layer delegating business logic to domain services.

The life-cycle of Domain Services and RESTful web services [2] is managed by different technologies. Domain services’ life-cycle is managed by OSGi; we don’t need to create instances of our domain services, OSGi will create them for us by scanning classes annotated with @Component, and they will be ready to be consumed if they are annotated with @Service (as illustrated in SwitchComponent.java—Sample Application OSGi Service Component—for more information see Providing Services with OSGi Declarative Services on page 159). In the other hand, RESTful web services are based on Servlets; Jersey Servlet [2] manages the life-cycle of the REST APIs. Similarly to Domain Services, we don’t need to create instances of our REST APIs, the Jersey Servlet will create them for us by scanning classes annotated with @Path; the Jersey Servlet handles HTTP requests and dispatches to the right REST API based on the @Path annotations (as illustrated in Creating Domain Service Resource (REST Interface of Business Logic Service) on page 169). The web container manages the life-cycle of the Jersey Servlet (as illustrated in Figure 3); the Jersey Servlet is defined at hm-rs/src/main/webapp/WEB-INF/web.xml.

Therefore, it is not possible to have OSGi injecting Domain Services into RESTful Web Services because their life-cycle is managed by different technologies: OSGi and Servlets respectively. In order to overcome this restriction and allow RESTful Web Services delegating to Domain Services the HP VAN SDN Controller Framework provides a Domain-Service Repository (ServiceLocator) that follows the Singleton Pattern [34]. However, it is necessary to write an OSGi compliant service that subscribes/unsubscribes our Domain Services to/from the repository. Create the ServiceAssistant class shown in the following listing under hm-rs module.

ServiceAssistant.java:

```java
package com.hp.hm.rs;

import org.apache.felix.scr.annotations.Component;
```

```java
```
import org.apache.felix.scr.annotations.Reference;
import org.apache.felix.scr.annotations.ReferenceCardinality;
import org.apache.felix.scr.annotations.ReferencePolicy;
import org.apache.felix.scr.annotations.References;
import com.hp.hm.api.SwitchService;
import com.hp.sdn.rs.misc.ServiceLocator;
...
@Component(immediate=true, specVersion="1.1")
@References(
    value=
    {// Add a @Reference (Separated by comma) for each
       // domain service exposed to the REST layer.
        @Reference(name="SwitchService",
                   referenceInterface = SwitchService.class,
                   policy=ReferencePolicy.DYNAMIC,
                   cardinality=ReferenceCardinality.OPTIONAL_MULTIPLE
        )
    }
)
public class ServiceAssistant {

    // Add a bind/unbind methods for each Domain Service
    // exposed to the REST layer.

    protected void bindSwitchService(SwitchService service,
                                      Map<String, Object> properties) {
        ServiceLocator.INSTANCE.register(SwitchService.class,
                                         service, properties);
    }

    protected void unbindSwitchService(SwitchService service) {
        ServiceLocator.INSTANCE.unregister(SwitchService.class, service);
    }
}

ServiceAssistant shows an alternative way of declaring dependencies. ServiceAssistant is annotated with @References instead of declaring a variable of type SwitchService and then annotate it with @Reference as in Consuming Services with OSGi Declarative Services on page 166 under the Dependent SwitchComponent.java listing. In this case we wouldn’t use the variable since we pass the bound service to the ServiceLocator.

The sample application’s domain service (SwitchService) is ready to be used by the REST layer. The following listing shows an extract of a modified SwitchResource (from Creating Domain Service Resource (REST Interface of Business Logic Service) on page 169) that makes use of the inherited get(Class<?>) method to get a reference to the SwitchService.

Consuming Domain Services:
    package com.hp.hm.rs;
import com.hp.hm.api.SwitchService;

@Path("switches")
public class SwitchResource extends ControllerResource {

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response list() {
        SwitchService service = get(SwitchService.class);
        List<Switch> switches = service.getAll();
        String result = "{switches:[]}"; // TODO: Encode switches
        return ok(result).build();
    }
}

The following SwitchResourceTest.java Mocking Domain Services listing shows an extract of a
modified SwitchResourceTest (from RESTful Web Services Unit Test on page 178) that uses
EasyMock[37] to mock SwitchService. Note how SwitchResourceTest registers the service mock
before the test and unregisters it after.

SwitchResourceTest.java Mocking Domain Services:

package com.hp.hm.rs;

import org.easymock.EasyMock;
import com.hp.hm.api.SwitchService;

public class SwitchResourceTest extends ControllerResourceTest {
    private static final String BASE_PATH = "switches";
    private SwitchService switchServiceMock;

    public SwitchResourceTest() {
        super("com.hp.hm.rs");
    }

    @Override
    @Before
    public void setUp() throws Exception {
        super.setUp();
        ResourceTest.setDefaultMediaType(MediaType.APPLICATION_JSON);

        switchServiceMock = EasyMock.createMock(SwitchService.class);
        sl.register(SwitchService.class, switchServiceMock,
                    Collections.<String, Object> emptyMap());
    }
}
@Override
@After
public void tearDown() throws Exception {
    super.tearDown();
    sl.unregister(SwitchService.class, switchServiceMock);
}

@Test
public void testList() {
    // Create mocks and define test case data
    List<Switch> switches = Collections.emptyList(); // Create test case
    // Recording phase (Define expectations)
    EasyMock.expect(switchServiceMock.getAll()).andReturn(switches);
    // Execution phase
    EasyMock.replay(switchServiceMock);
    String response = get(BASE_PATH);
    // Verification phase
    String expectedResponse = "{"switches":[]}");
    assertResponseContains(response, expectedResponse);
    EasyMock.verify(switchServiceMock);
}


JSON Encoding

As described previously, the tasks a REST API normally accomplishes is decoding the request and encoding the result into the response. This sample application uses JSON [36] format but could have used any other, like XML. There are several different tools to assist on JSON conversion and any tool and any way of organizing the codecs (or converters) could have been selected. However, the HP VAN SDN Controller SDK offers some infrastructure classes and services with the aim of unifying the way JSON codecs are implemented and shared. The generated sample application is too simple to use a codec, so a more complex example using a codec is presented here. This example makes use of such JSON API to implement a JSON codec for the Switch model object so it is used by the SwitchResource.

Implementation of the JSON codecs is located at the hp-rcs module; however for real applications creating a new module to locate codecs might result in a better organization. The listing, SwitchJsonCodec.java, shows the JSON codec for Switch (Defining Model Objects on page 150). SwitchJsonCodec.java:
package com.hp.hm.rs.json;

import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.node.ObjectNode;
import com.hp.util.json.AbstractJsonCodec;
import com.hp.util.json.JsonCodec;
...

public class SwitchJsonCodec extends AbstractJsonCodec<Switch> {

    private static final String ID = "id";
    private static final String MAC_ADDRESS = "mac_address";
    private static final String IP_ADDRESS = "ip_address";
    private static final String FRIENDLY_NAME = "friendly_name";
    private static final String ACTIVE_STATE = "active_state";

    public SwitchJsonCodec() {
        super("switch", "switches");
    }

    @Override
    public Switch decode(ObjectNode node) {
        validateMandatoryFields(node, MAC_ADDRESS);

        MacAddress macAddress = MacAddress.valueOf(
            node.get(MAC_ADDRESS).asText());

        Id<Switch, Long> id = null;
        if (!node.path(ID).isMissingNode()) {
            id = Id.valueOf(Long.valueOf(node.get(ID).asLong()));
        }

        Switch device = new Switch(id, macAddress);

        if (!node.path(IP_ADDRESS).isMissingNode()) {
            device.setIpAddress(IpAddress
                .valueOf(node.get(IP_ADDRESS).asText()));
        }

        if (!node.path(FRIENDLY_NAME).isMissingNode()) {
            device.setFriendlyName(node.get(FRIENDLY_NAME).asText());
        }

        if (!node.path(ACTIVE_STATE).isMissingNode()) {
            device.setActiveState(ActiveState.valueOf(
                node.get(ACTIVE_STATE).asText()));
        }

        return device;
    }
}
return device;
}

@override
public ObjectNode encode(Switch device) {
    ObjectNode node = mapper.createObjectNode();

    node.put(MAC_ADDRESS, device.getMacAddress().toString());

    if (device.getId() != null) {
        node.put(ID, device.getId().getValue().longValue());
    }

    if (device.getIpAddress() != null) {
        node.put(IP_ADDRESS, device.getIpAddress().toString());
    }

    if (device.getFriendlyName() != null) {
        node.put(FRIENDLY_NAME, device.getFriendlyName());
    }

    if (device.getActiveState() != null) {
        node.put(ACTIVE_STATE, device.getActiveState().name());
    }

    return node;
}

private static void validateMandatoryFields(ObjectNode node,
                                               String... fields) throws IllegalArgumentException {
    if (fields != null) {
        for (String field : fields) {
            if (node.path(field).isMissingNode()) {
                throw new IllegalArgumentException("JSON node '" + node
                                + '" is missing field '" + field + '"");
            }
        }
    }
}

There are some dependencies to declare in order to implement the codecs. Open the hmr
rs/pom.xml file and add the XML extract from the JSON Module Dependencies listing to the
<dependencies> node; after updating the POM file update the Eclipse project dependencies (see Updating Project Dependencies on page 146).
The HP VAN SDN Controller SDK uses the Jackson API [38] as the underlying API to handle JSON conversion.

In order to make the `SwitchJsonCodec`, and any other codec in the application, available so it can be reused, create a JSON factory that is registered to the central JSON repository which is exposed as a regular service called `JsonService`. The following listing shows the implementation of this JSON factory:

**HmJsonFactory.java:**
```java
package com.hp.hm.rs.json;

import com.hp.util.json.AbstractJsonFactory;
import com.hp.util.json.JsonFactory;
...
@Component
@Service
@Property(name = "app", value = "flare")
public class HmJsonFactory extends AbstractJsonFactory {

    public HmJsonFactory() {
        // Register all application's JSON codecs
        addCodecs(Switch.class, new SwitchJsonCodec());
    }

    @Deactivate
    protected void deactivate() {
        clearCodecs();
    }
}
```

*HmJsonFactory* holds all the JSON codecs; it is an OSGi service so it is registered to the central JSON repository when it is activated and unregistered from the JSON repository when it is deactivated. The registration happens automatically because the HP VAN SDN Controller Framework observes all activated JSON Factory (`JsonFactory`) services annotated with the following property: “name=flare”.

---

186
Now that the JSON factory is in place, update the `SwitchResource` to use the `JsonService` to encode and decode `Switch` objects. The `SwitchResource.java` Using JSON Codecs listing shows a modification of `SwitchResource` that uses a JSON codec to encode `Switch` objects.

`SwitchResource.java` Using JSON Codecs:
```java
package com.hp.hm.rs;

import com.hp.sdn.json.JsonService;
...
@Path("switches")
public class SwitchResource extends ControllerResource {

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response list() {
        SwitchService service = get(SwitchService.class);
        List<Switch> switches = service.find(null, null);
        JsonService jsonService = get(JsonService.class);
        String result = jsonService.toJsonList(switches, Switch.class, true);
        return ok(result).build();
    }
    ...
}
```

`JsonService Module Dependencies:`
```xml
<dependency>
    <groupId>com.hp.sdn</groupId>
    <artifactId>sdn-cmmon-api</artifactId>
    <version>${sdn.version}</version>
</dependency>
```

The following listing, `SwitchResourceTest.java` Using JSON Codecs, is a modification of the `SwitchResourceTest.java` that also uses `JsonService` to complete the tests.

`SwitchResourceTest.java` Using JSON Codecs:
```java
package com.hp.hm.rs;

import com.hp.sdn.json.JsonService;
...
public class SwitchResourceTest extends ControllerResourceTest {

    private final String BASE_PATH = "switches";

    private SwitchService switchServiceMock;
    private JsonService jsonServiceMock;

    public SwitchResourceTest() {
```
super("com.hp.hm.rs");
}

@Suppress
@Override
@Before
public void setUp() throws Exception {
    super.setUp();
    ResourceTest.setDefaultMediaType(MediaType.APPLICATION_JSON);
    switchServiceMock = EasyMock.createMock(SwitchService.class);
    sl.register(SwitchService.class, switchServiceMock,
                Collections.<String, Object> emptyMap());
    jsonServiceMock = EasyMock.createMock(JsonService.class);
    sl.register(JsonService.class, jsonServiceMock,
                Collections.<String, Object> emptyMap());
}

@Override
@After
public void tearDown() throws Exception {
    super.tearDown();
    sl.unregister(SwitchService.class, switchServiceMock);
    sl.unregister(JsonService.class, jsonServiceMock);
}

@Test
public void testList() {
    // Create mocks and define test case data
    List<Switch> switches = Collections.emptyList();
    // Note that the expected switches can be anything
    // (it doesn't matter) since the SwitchService has been mocked.
    String switchesJson = "{""switches"":[]}";
    // Note that the returned JSON does not matter since
    // the JSON codec has been mocked.
    // Recording phase (Define expectations)
    EasyMock.expect(switchServiceMock.find(EasyMock.isNull(SwitchFilter.class),
                                             EasyMock.isNull(SortSpecification.class))).andReturn(switches);
    EasyMock.expect(jsonServiceMock.toJsonList(EasyMock.same(switches), EasyMock.eq(Switch.class),
                                                EasyMock.eq(true))).andReturn(switchesJson);
}
// Execution phase

EasyMock.replay(switchServiceMock, jsonServiceMock);
String response = get(BASE_PATH);

// Verification phase

assertResponseContains(response, switchesJson);
EasyMock.verify(switchServiceMock, jsonServiceMock);

...
Controller-Controller Communication via REST (Sideways APIs)

RESTful Web Services (or REST APIs) \([1] [2]\) also represent a convenient way to enable communication between controllers, and the HP VAN SDN Controller framework provides some facilities to do so. This section illustrates a way to enable such communication. This section is optional and the code illustrated here won’t be part of our sample application, it is just a section dedicated to illustrate this useful communication mechanism. Also note this should not be the preferred mechanism to enable communication between controllers, the HP VAN SDN Controller Framework offers other services based on Hazelcast \([39]\) to achieve that. For more information see Distributed Coordination Service on page 67.

Figure 55 illustrates the intuitive idea. In order to enable communication, a new service in charge of the communication is created to decouple the business logic from the specifics of the underlying communication technology. The implementation of the communication service sends HTTP requests to the destination REST Web Service and processed HTTP responses. By introducing this communication service it is possible to define higher-level (type-safe) communication methods.

![Controller-Controller Communication via REST (Sideway API)](image)

For example assume there is a need to retrieve all the open flow switches controlled by a remote system. A sideway (or transfer) API could be defined to take care of such communication as shown in the following listing.

SwitchTransferService.java:
```java
package com.hp.hm.api;

public interface SwitchTransferService {
    public Set<Switch> getControlledDevices(IpAddress system);
}
```

The following listing shows an extract of the communication service implementation using the facilities provided by the HP VAN SDN Controller framework to handle HTTP requests and responses.

SwitchTransferManager.java:
```java
package com.hp.hm.impl;

import javax.ws.rs.core.Response.Status;
import com.hp.hm.api.SwitchTransferService;
```
import com.hp.sdn.json.JsonService;
import com.hp.sdn.misc.ResponseData;
import com.hp.sdn.misc.ServiceRest;
import com.hp.util.StringUtils;
...
@Component
@Service
public class SwitchTransferManager implements SwitchTransferService {

    // Some specific dependencies (like javax.ws.rs.core.Response) are needed
    // to implement transfer services that use RESTful web services as the
    // underlying mechanism to achieve communication. It is recommended to
    // locate transfer services in a separated module.

    static final String BASE_DESTINATION_PATH = "sdn/hm/v1.0/switches";

    @Reference(policy = ReferencePolicy.DYNAMIC,
                cardinality = ReferenceCardinality.MANDATORY_UNARY)
    private volatile ServiceRest restClient;

    @Reference(policy = ReferencePolicy.DYNAMIC,
                cardinality = ReferenceCardinality.MANDATORY_UNARY)
    private volatile JsonService jsonService;

    @Override
    public Set<Switch> getControlledDevices(IpAddress ipAddress) {

        URI uri = restClient.uri(ipAddress, BASE_DESTINATION_PATH);
        responseData = restClient.get(restClient.login(), uri);

        try {
            responseData = new String(response.data(), StringUtils.UTF8);
        } catch (UnsupportedEncodingException e) {
            throw new RuntimeException("Unable to decode response from " + ipAddress, e);
        }

        if (response.status() != Status.OK.getStatusCode()) {
            StringBuilder message = new StringBuilder(32);
            message.append("Unable to communicate with ");
            message.append(ipAddress);
            message.append(". Status code: ");
            message.append(response.status());
            message.append(". Response data: ");
            message.append(responseData);

            throw new RuntimeException("Unable to decode response from " + ipAddress, e);
        }
    }
}
throw new RuntimeException(message.toString());
}

List<Switch> remoteDevices = jsonService.fromJsonList(responseData, Switch.class);
return new HashSet<Switch>(remoteDevices);
}

ServiceRest is a service provided by the HP VAN SDN Controller framework that enables HTTP communication by offering the common operations GET, POST, PUT and DELETE. It also takes care of service authentication. In order to use ServiceRest we need to add the module it is located at as a dependency. Open the `hm-bl/pom.xml` file and add the XML extract from the following listing, ServiceRest Dependency, to the `<dependencies>` node; after updating the POM file update the Eclipse project dependencies (see Updating Project Dependencies on page 146).

ServiceRest Dependency:

```xml
<dependency>
    <groupId>com.hp.sdn</groupId>
    <artifactId>sdn-common-api</artifactId>
    <version>${sdn.version}</version>
</dependency>
<dependency>
    <groupId>com.hp.sdn</groupId>
    <artifactId>sdn-common-misc</artifactId>
    <version>${sdn.version}</version>
</dependency>
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-server</artifactId>
    <version>1.17</version>
    <scope>compile</scope>
</dependency>
<dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
    <version>4.2.1</version>
    <scope>test</scope>
</dependency>
```

In order to use `SwitchTransferService`, inject a reference into the business logic implementation (SwitchManager for example) as depicted in Consuming Services with OSGi Declarative Services on page 166. Note how SwitchTransferManager with @Component and @Service is directly annotated. It is possible to have followed the same pattern described in Providing Services with OSGi Declarative Services on page 159 to keep our communication service implementation clean from the OSGi restrictions, however communication service implementations rarely consume other services and thus there is no need of dealing with the fact that dependency components may come and go (Binding/unbinding injected references).
In real applications creating `new modules to locate communication services would result in a better organization: For example, using `hm-ext-api module for the communication service interfaces (instead of `hm-api as in this example) and `hm-ext for the implementations (instead than `hm-bl as in this example).

### Creating RSdoc

Trying the REST API with Curl on page 177 describes a way to try the REST API by executing commands. The HP VAN SDN Controller SDK offers a method to create a semi-automated interactive RESTful API documentation which offers a better way to interact with REST APIs. It is called RSdoc because is a combination of JAX-RS [2] and Javadoc [21].

One big advantage of RSdoc is that JAX-RS annotations and Javadoc are already written when implementing RESTful Web Services, thus in order to enable the application to create the RSdoc is relatively easy and automatic: a few configuration files need to be updated.

Create a JSON [36] schema to express the data model. Create `hm-rs/src/main/resources/model.json file with the content from the following RSdoc JSON Schema listing. (To add more schemas, separate them by comma).

RSdoc JSON Schema:

```json
{
  "com.hp.hm.model.Switch":
  {
    "properties":
    {
      "id": {"type": "long"},
      "mac_address": {"type": "string"},
      "ip_address": {"type": "string"},
      "name": {"type": "string"},
      "active_state": {"type": "string"}
    }
  }
}
```

Create a class to register the REST API documentation provider under `hm-rs module as in the following `DocProvider.java listing. In order to extend from `SelfRegisteringRSDocProvider a dependency must be added. Open the `hm-rs/pom.xml file and add the XML extract from the RSdoc Provider Dependency listing to the `<dependencies>` node; after updating the POM file update the Eclipse project dependencies (see Updating Project Dependencies on page 146).

`DocProvider.java`:

```java
package com.hp.hm.rs;

import org.apache.felix.scr.annotations.Component;
import com.hp.sdn.adm.rsdoc.RSDocProvider;
import com.hp.sdn.adm.rsdoc.SelfRegisteringRSDocProvider;

@Component
public class DocProvider extends SelfRegisteringRSDocProvider {
```
public DocProvider() {
    super("hm", "rsdoc", DocProvider.class.getClassLoader());
}

NOTE
The name used to call the super class constructor ("hm" in DocProvider.java listing) must not contain spaces; it may be any name but with no spaces because it is used to generate internal paths.

RSdoc Provider Dependency:

```xml
<dependency>
    <groupId>com.hp.sdn</groupId>
    <artifactId>sdn-adm-api</artifactId>
    <version>${sdn.version}</version>
</dependency>
```

Modify hm-rs/pom.xml file to include the plug-in in charge of executing the command used to generate the RSdoc as shown in the following RSdoc Generation Maven Configuration listing. This plugin executes a tool offered by the HP VAN SDN Controller SDK that generates the RSdoc based on the parameters used in RSdoc Generation Maven Configuration listing.

RSdoc Generation Maven Configuration:

```xml
...<properties>
    <banned.rs.paths>com.hp.hm.rs</banned.rs.paths>
    <webapp.context>sdn/hm/v1.0</webapp.context>
    <web.context.path>sdn/hm/v1.0</web.context.path>

    <!-- RSdoc properties -->
    <api.name>Device Health Monitor v1.0</api.name>
    <api.version>1.0</api.version>
    <api.url>https://localhost:8443/${webapp.context}</api.url>
</properties>
...
<build>
    <plugins>
        <plugin>
            <artifactId>maven-antrun-plugin</artifactId>
            <executions>
                <execution>
                    <id>generate-resources</id>
                    <phase>process-resources</phase>
                    <configuration>
                        <tasks>
                            <delete dir="target/classes/rsdoc" />
                            <mkdir dir="target/classes/rsdoc" />
                        </tasks>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
...
Build and install the application as described Building the Application on page 146 and Installing the Application on page 147. RSdoc is now accessible as illustrated in Figure 56.

Figure 56 Sample Application RSdoc

If for some reason you don’t want a RESTful web service (method annotated with a REST verb: @GET, @POST, @PUT, @DELETE) to appear in the RSdoc - maybe because it is not ready for
consumption or because it is meant to be used internally by a sideway API (see Controller-Controller Communication via REST (Sideways APIs) on page 190) it may be annotated with\
@RsDocIgnore as illustrated in the following listing.

RsDocIgnore Annotation:

```java
package com.hp.hm.rs;

import com.hp.api.rsdoc.RsDocIgnore;
...
@Path("mypath")
public class MyResource extends ControllerResource {

    @GET
    @Path("internal")
    @Produces(MediaType.APPLICATION_JSON)
    @RsDocIgnore
    public Response internalMethod() {
        ...
    }
}
```

### Trying the REST API with RSdoc

At this point try the REST API using the RSdoc, which is the preferred method. Follow the steps from Rsdoc Live Reference on page 17 to open Rsdoc and authenticate, and then try the sample application’s REST API as illustrated in Figure 57. Modify SwitchManager to return some fake data in the getAll method and try GET switches from the RSdoc.
NOTE

The tool offered by the HP VAN SDN Controller SDK that generates the RSdoc takes the Javadoc [21] to generate the REST API documentation as illustrated in Figure 11. Therefore, it is mandatory to write Javadoc for the REST APIs (In general, production code classes should be properly documented). If a REST API method does not contain Javadoc, the entire REST API won’t be included in the RSdoc.

Creating a GUI

The following information describes the process of creating user interfaces using the HP SKI framework and integrating such views to the HP VAN SDN Controller GUI. For more information see GUI on page 59.
Creating Views

The SKI framework uses JavaScript [40] as the underlying technology, thus the views are Dynamic-HTML based. Start by creating the application’s cascading style sheets [41]. Create the file *hm-ui/src/main/webapp/css/hm.css* with the content from the following *hm.css* listing. This example uses a very simple cascading style sheet, however any style desired can be created and as many style sheets as needed.

hm.css:

```
.hm {
  background-color: red;
}
```

Now create a view to display the Open Flow Switches. This example shows a tool bar button that updates the view’s content with the “Hello World” message when it is pressed (see SKI Framework - Overview on page 59 to find a SKI reference application that provides examples of SKI widgets). Create the file *hm-ui/src/main/webapp/js/hm.js* with the content from the following listing, *hm.js*. Create one JavaScript file for each view in the application.

hm.js:

```
// JSLint directive...
/*global $: false*/
(function (api) {
  'use strict';

  //framework APIs
  var f = api.fn, //general API
      def = api.def, //application definition API
      v = api.view; //view API

  f.trace('including hm.js');

  // Create a view with a toolbar button
  function load(view) {
    v.setToolbar(def.tbButton(view.mkId('btn'),
      view.lion('button'), '', function () {
        $.get('/sdn/ui/switches/app/rs/hm', function(data) {
          v.setContent($('<span/>'). append(data));
        });
      }));
  }

  def.addView('hm', {
    load: load
  });

  def.insertViewsAfter('exportLogs',
```
def.addView('hmTab')
}

{(SKI)};

Now create a script that adds a menu entry to the navigation panel so the `hm` view is accessible from the SDN Controller’s GUI. Create the file `hm-ui/src/main/webapp/js/hm-nav.js` with the content from the following `hm-nav.js` listing. Use `hm-nav.js` to add as many entries in the navigation panel as the application needs; there is just one JavaScript file dealing with the navigation panel.

`hm-nav.js`:

```javascript
// JSLint directive...
/*global $: false, SKI: false */
(function (api) {
  'use strict';

  var f = api.fn,  // general functions API
       nav = api.nav;  // navigation model API

  f.trace('including hm-nav.js');

  // Adds a new category and a new item
  nav.insertCategoryAfter('c-tasks', 'c-hm', [
    nav.item('n-hm-task', 'hmTask', 'square')
  ]);  

  // Add a new item to an existing category
  nav.insertItemsBefore('n-exportLogs', {
    nav.item('n-hm-task', 'hmTask', 'square')
  });

})(SKI);
```

Plain text is not used in the previous two listing. In order to display text in the views define properties files which contain text identified by keys. This allows for localizing the applications; see GUI on page 59 for details on how the localization infrastructure works on the SKI framework.

Next, create the properties files to define the text that the previous two listing reference by their key. Create one properties file for each view.

When adding the “Switches” view to the framework in `hm.js`, name it “hm” (Highlighted in the `hm.js` listing), thus the associated properties file must have the same name. Create the file `hm-ui/src/main/resources/com/hp/hm/ui/lion/hm.properties` with the content from the following `hm.properties` listing. Title and icon keys are reserved keys automatically used by the framework to set the view’s title and icon. For more information see GUI on page 59 for details about available icons and how define custom icons.

`hm.properties`:

```properties
title = Health Monitor Title
```
On the other hand, the properties file associated to `hm-nav.js` from the `hm-nav.js` listing needs a special treatment: `nav-lion.properties` is a reserved name for properties files that contain text associated to the navigation panel, and since `hm-nav.js` is adding content to it, add the text in the file `hm-ui/src/main/resources/com/hp/hm/io/lion/nav-lion.properties`. The following `nav-lion.properties` listing shows the content of the `nav-lion.properties` file.

```
nav-lion.properties:
    # Navigation category: Health Monitor
    c-hm = Health Monitor
    n-hm = Health Monitor Things
    n-hm-task = Health Monitor Task
```

**Integrating Views to the SDN Controller GUI**

In order to integrate views to the HP VAN SDN Controller an UI extension needs to be registered so the framework hooks the views into the controller’s GUI. The SDN Controller SDK provides a `SelfRegisteringUIExtension` class that can be used to subscribe the application’s views. The following `UIExtension.java` listing illustrates the way of subscribing the user interface.

```
UIExtension.java:
    package com.hp.hm.ui;

    import org.apache.felix.scr.annotations.Component;
    import com.hp.sdn.ui.misc.SelfRegisteringUIExtension;

    @Component
    public class UIExtension extends SelfRegisteringUIExtension {

        public UIExtension() {
            super("hm", "com/hp/hm/ui", UIExtension.class);
        }
    }
```

Some dependencies need to be added so `SelfRegisteringUIExtension` can be used; open the `hm-ui/pom.xml` file and add the XML extract from the following `UIExtension Module Dependencies` listing to the `<dependencies>` node; after updating the POM file update the Eclipse project dependencies (see Updating Project Dependencies on page 146).

**UIExtension Module Dependencies:**

```
<dependency>
    <groupId>com.hp.sdn</groupId>
    <artifactId>sdn-adm-rs-misc</artifactId>
    <version>${sdn.version}</version>
</dependency>
<dependency>
    <groupId>com.hp.sdn</groupId>
    <artifactId>sdn-adm-rs-misc</artifactId>
    <version>${sdn.version}</version>
</dependency>
```
The second parameter ("com/hp/hm/ui") of the super(…) call in the constructor of the UIExtension.java listing specifies the location of the two files the framework uses to integrate the views: css.html and js.html. These files act as links to the application’s cascading style sheets [41] and the application’s JavaScript files respectively. Create the files hm-ui/src/main/resources/com/hp/hm/ui/css.html and hm-ui/src/main/resources/com/hp/hm/ui/js.html with the content from the following UIExtension css.html listing and the following UIExtension js.html listing.

UIExtension css.html:

```html
<link href="/sdn/ui/health/css/hm.css" rel="stylesheet">
```

UIExtension js.html:

```html
<script src="/sdn/ui/health/js/hm-nav.js"></script>
<script src="/sdn/ui/health/js/hm.js"></script>
<script src="/sdn/ui/health/app/hmChart.js"></script>
```

The prefix "sdn/ui/health" used in the UIExtension css.html listing and the UIExtension js.html listing must match the web.context.path property from the hm-ui/pom.xml to generate .war listing under Module Configuration below. The rest of the path ("…/css/hm.css", "…/js/hm-nav.js" and "…/js/hm.js") is determined by the structure of the hm-ui/src/main/webapp directory.

### Module Configuration

The SDN Controller GUI is based on the HP SKI framework which uses JavaScript [40] as the underlying technology. Thus, similarly to the RESTful web services module (hm-rs), the user interface module (hm-ui) is deployed as a web application; and thus the module’s output must be a web application archive (.war file) [35].

This section describes the configuration changes needed to do to the sample application so the hm-ui module is properly deployed.

Create a place holder hm-ui/src/main/webapp/WEB-INF/web.xml file with the content from the following UI Module Web Application (web.xml) listing.

UI Module Web Application (web.xml):

```xml
<?xml version="1.0" encoding="UTF-8"?>
  <display-name>Health Monitor UI</display-name>  . . .
</web-app>
```
Now update the **hm-ui** module POM file **hm-ui/pom.xml** with the extract shown in the following **hm-ui/pom.xml** to generate .war listing, to generate the .war file during the build process.

**hm-ui/pom.xml to generate .war:**

```xml
...<modelVersion>4.0.0</modelVersion>
<artifactId>hm-ui</artifactId>
<packaging>war</packaging>
...
<properties>
  <jersey.version>1.17</jersey.version>
  <banned.rs.paths>com.hp.hm.ui</banned.rs.paths>
  <webapp.context>sdn/ui/health</webapp.context>
  <web.context.path>sdn/ui/health</web.context.path>
</properties>
<dependencies>
...
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.felix</groupId>
      <artifactId>maven-bundle-plugin</artifactId>
      <version>2.3.6</version>
      <extensions>true</extensions>
      <executions>
        <execution>
          <id>bundle-manifest</id>
          <phase>process-classes</phase>
          <goals>
            <goal>manifest</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
  <manifestLocation>${project.build.directory}/ META-INF</manifestLocation>
  <supportedProjectTypes>
    <supportedProjectType>bundle</supportedProjectType>
    <supportedProjectType>war</supportedProjectType>
  </supportedProjectTypes>
  <instructions>
    <Import-Package>
      com.sun.jersey.api.core,
      com.sun.jersey.spi.container.servlet,
      com.sun.jersey.server.impl.container.servlet,
      com.hp.util.rs,
    </Import-Package>
  </instructions>
...
Next, update the application deployment plan `hm-app/hm.plan` to deploy the `hm-ui` module as illustrated in the following listing.

**Sample Application Deployment Plan Considering UI Module:**

Now try the application’s user interface. Build and install the application as described in Building the Application on page 146 and Installing the Application on page 147. After installing the application refresh the SDN Controller GUI as illustrated at the top part of Figure 58; the application’s GUI entry appears as illustrated at the bottom part of Figure 58. Figure 59 shows the application’s view after clicking the “Refresh Data” button.
GUI-Specific REST API

As seen previously the SKI framework uses JavaScript [40] as the underlying technology to create Dynamic-HTML based views. Such dynamism comes from logic executed at the SDN Controller or WEB server from the JavaScript point of view. The SKI framework integrates the jQuery [42] tool which allows for the execution of asynchronous HTTP requests. jQuery encapsulates AJAX [43] to achieve asynchronous calls: AJAX is the art of exchanging data with a server, and updating parts of a web page, without reloading the whole page.

Use jQuery to connect to the server to retrieve information via HTTP request and HTTP responses. RESTful web services [2] [1] was inspired by HTTP; as a result, REST can be used wherever HTTP can. A RESTful web API (also called a RESTful web service) is a web API implemented using HTTP and REST principles. Thus, use REST APIs to attend requests coming from the user interface.

The sample application already contains a module (hm-rs) for the RESTful Web Services that expose to the outside world functionality provided by the sample application; however this
functionality refers to the application’s domain model or business logic. Besides the domain model functionality, a view normally has requirements that are specific to presentation logic (For example a view could call the server to retrieve a catalog of pictures related to an item). It is not desired to pollute the RESTful web services from hm-rs module with presentation logic specific methods. Therefore, it’s considered a good practice creating GUI-specific REST APIs in the hm-ui module.

Similarly to the hm-rs module, in order to implement REST web services the module needs to declare some dependencies; open the hm-ui/pom.xml file and add the XML extract from the Creating Domain Service Resource (REST Interface of Business Logic Service) on page 169 under the “REST Module Dependencies” listing and the RESTful Web Services Unit Test on page 178 under “Resource Test Dependencies” listing, to the <dependencies> node (Remove any duplicates). After updating the POM file update the Eclipse project dependencies (see Updating Project Dependencies on page 146).

The following SwitchResource.java listing shows how to create the Switch View REST API named SwitchResource. The SwitchResource.java listing shows an extract of the resource. To use JSON encoding see JSON Encoding on page 183. To write unit test follow the instructions from RESTful Web Services Unit Test on page 178.

SwitchResource.java:

```java
package com.hp.hm.ui.rs;
...
@Path("hm")
public class SwitchResource extends ControllerResource {

@GET
@Produces(MediaType.TEXT_PLAIN)
public Response hello() {
    return ok("The world is all about Switch!!! <p>" + "The Hewlett-Packard is here to prove it by providing" + "you with a Test APP to verify SDK guide, the Health Monitor").build();
}
}
```

Now update the web.xml placeholder added in Module Configuration on page 201 under the “UI Module Web Application (web.xml)” listing. The Jersey Servlet [2] that handles HTTP requests and dispatches to the right REST API based on the @Path annotations needs to be defined. Update hm-ui/src/main/webapp/WEB-INF/web.xml file with the content from the following listing:

UI Module Web Application (web.xml) Defining Jersey Servlet:

```xml
<?xml version="1.0" encoding="UTF-8"?>
  <display-name>Health Monitor UI</display-name>
  ...
  <servlet>
```
<servlet-name>GUI REST Services</servlet-name>
<servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>

<!-- Authentication Filter -->
<init-param>
  <param-name>com.sun.jersey.spi.container.ContainerRequestFilters</param-name>
  <param-value>com.hp.util.rs.auth.AuthJerseyFilter</param-value>
</init-param>

<init-param>
  <param-name>exclude-paths</param-name>
  <param-value>^$</param-value>
</init-param>

<init-param>
  <param-name>com.sun.jersey.config.property.resourceConfigClass</param-name>
  <param-value>com.sun.jersey.api.core.ClassNamesResourceConfig</param-value>
</init-param>

<init-param>
  <param-name>com.sun.jersey.config.property.classnames</param-name>
  <param-value>
    <!-- Application REST API -->
    com.hp.hm.ui.rs.SwitchResource
    <!-- Application Error Handlers -->
    <!-- Provided Error Handlers -->
    com.hp.sdn.rs.misc.DuplicateIdErrorHandler
    com.hp.sdn.rs.misc.NotFoundErrorHandler
    com.hp.sdn.rs.misc.ServiceNotFoundErrorHandler
    com.hp.sdn.rs.misc.IllegalDataHandler
    com.hp.sdn.rs.misc.IllegalStateHandler
    com.hp.sdn.rs.misc.AuthenticationHandler
  </param-value>
</init-param>

</servlet>

<servlet-mapping>
  <servlet-name>GUI REST Services</servlet-name>
  <url-pattern>/app/rs/*</url-pattern>
</servlet-mapping>

<filter>
  <filter-name>Gui Jersey Filter</filter-name>
  <filter-class>com.sun.jersey.spi.container.servlet.ServletContainer</filter-class>
</filter>
<filter-name>Token Authentication Filter</filter-name>
<filter-class>com.hp.sdn.rs.misc.TokenAuthFilter</filter-class>
</filter>

<filter-mapping>
<filter-name>Token Authentication Filter</filter-name>
<url-pattern>/app/rs/*</url-pattern>
</filter-mapping>
</web-app>

Now update the switches view from Creating Views on page 198 under the “hm.js” listing to make
the remote call as illustrated in the following listing:

hm.js Requesting Data to the Controller:

...  
// Create a view with a toolbar button
function load(view) {
  v.setToolbar(def.tbButton(view.mkId('btn'), view.lion('button'), '',
  function () {
    $.get('/sdn/ui/health/app/rs/hm', function(data) {
      v.setContent($('<span/>').append(data));
    });
  });

}  
...

As seen in the code the view connects to the relative path “/sdn/ui/health/app/rs/hm” so the
connection is opened to the same controller that generated the web page. The prefix “/sdn/ui/health” must match the web.context.path property defined in Module Configuration on
page 201 under the “hm-ui/pom.xml to generate .war” listing. The infix “app/rs” is given by the
Jersey Servlet mapping configuration in GUI-Specific REST API on page 204 under the “UI Module
Web Application (web.xml) Defining Jersey Servlet” listing, and the suffix “hm” is the relative path
of the resource given by the @Path annotation in the GUI-Specific REST API on page 204 under
the “SwitchResource.java” listing.

In this case, the response media type (in SwitchResource) is defined as TEXT_PLAN. This means
that the data parameter of the $.get() function callback is filled in with a plain string. The media
type can also be defined as APPLICATION_JSON and return a JSON formatted string. In which
case, the JavaScript [40] data parameter would be an object.

No need to worry about authentication because the SKI framework automatically includes the
authentication token generated after login (Figure 40) into the HTTP request headers.

Now try the application’s user interface again. Build and install the application as described in
Building the Application on page 146 and Installing the Application on page 147. After installing
the application refresh the SDN Controller GUI as illustrated at the top part of Figure 58; the
application’s GUI entry will appear as illustrated at the bottom part of Figure 58. Now the
message returned by SwitchResource can be seen after clicking the “Refresh Data” button.
Using SDN Controller Services

In the following information some of the services provided by the HP VAN SDN Controller will be consumed to illustrate the philosophy followed by the controller: OSGi declarative services as depicted in section Consuming Services with OSGi Declarative Services on page 166.

Services published by the controller are meant to be consumed by the application’s business logic. Some services are available to the RESTful web services, however, as depicted in Domain Service - REST API Integration on page 180, web services should not implement any logic but controller logic. Thus, it is considered a good practice to always delegate to the business logic.

At this point RESTful web services and business logic are fully integrated. A simple in-memory data structure will be used to store OpenFlow switches data. The following listings illustrate the complete implementation of SwitchManager and SwitchResource that will be used to consume HP VAN SDN Controller services. These implementations allow the REST API to be functional for small transient data (Filtering and sorting still pending).

NOTE

Synchronization on the in-memory data structure and the fact that Switch is a mutable class have been intentionally ignored. Even though it is important to consider the multi-threaded environment nature of RESTful web services and data protection (since the same references from the in-memory data store are returned by the business logic) they are irrelevant for the purpose of the illustration: Consuming services published by the controller. A more serious implementation would make use of the synchronization tools offered by Java and make copies of the objects before they are returned (Adding a copy constructor in Switch for example) or even better, use a database. Complicated code is avoided for illustration purposes.

SwitchManager.java In-Memory Data Storage:

```java
package com.hp.hm.impl;
...
public class SwitchManager implements SwitchService {

@SuppressWarnings("unused")
private final SystemInformationService systemInformationService;

@SuppressWarnings("unused")
private AlertService alertService;

private Map<Id<Switch, UUID>, Switch> devices;
private AtomicLong idCount;

public SwitchManager(SystemInformationService systemInformationService) {
    if (systemInformationService == null) {
        throw new NullPointerException(...);
    }
    this.systemInformationService = systemInformationService;
}
```

208
devices = new HashMap<Id<Switch, UUID>, Switch>();
idCount = new AtomicLong(1);

public void setAlertService(AlertService alertService) {
    this.alertService = alertService;
}

@Override
public Switch create(String name) {
    Switch device = new Switch(name);
    if (isEmpty(device.name())) {
        device.setName("Switch-" + device.getId().getValue().toString());
    }
    devices.put(device.getId(), device);
    return device;
}

@Override
public Collection<Switch> getAll() {
    synchronized (devices) {
        return Collections.unmodifiableCollection(devices.values());
    }
}

@Override
public Switch get(Id<Switch, UUID> id) {
    if (id == null) {
        throw new NullPointerException("id cannot be null");
    }
    synchronized (devices) {
        Switch s = devices.get(id);
        if (s == null)
            throw new NotFoundException("Switch with id " + id + " not found");
        return s;
    }
}

@Override
public void delete(Id<Switch, UUID> id) {
    if (id == null) {

throw new NullPointerException("id cannot be null");
}
synchronized (devices) {
    Switch s = devices.remove(id);
    if (s == null) {
        throw new NotFoundException("Switch with id " + id + " not found");
    }
}

SwitchResource.java Delegating to Business Logic:
package com.hp.hm.rs;
...
@Path("health")
public class SwitchResource extends ControllerResource {

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response getAll() {
        SwitchService service = get(SwitchService.class);
        ObjectMapper mapper = new ObjectMapper();
        ObjectNode root = mapper.createObjectNode();
        List<Switch> switches = service.getAll();
        for (Switch s: switches) {
            nodes.add(json(s, mapper));
            ArrayNode rowNode = root.putArray("health");
            rowNode.addAll(nodes);
        }

        return ok(root.toString()).build();
    }

    @GET
    @Path("/{uid}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response get(@PathParam("uid") long uid) {
        Id<Switch, UUID> deviceId = Id.valueOf(UUID.fromString(uid));

        SwitchService service = get(SwitchService.class);
        Switch device = service.get(deviceId);

        return response(device, new ObjectMapper()).build();
    }
}
@POST
@Produces(MediaType.APPLICATION_JSON)
public Response create(String request) {ObjectMapper mapper = new
ObjectMapper();

JsonNode root = parse(mapper, request, "Switch data");
JsonNode node = root.path("item");

String name = exists(node, "name") ? node.path("name").asText() : null;

SwitchService service = get(SwitchService.class);
Switchdevice = service.create(name);
return response(device, mapper).build();
}

@DELETE
@Path("{uid}")
@Produces(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response delete(@PathParam("uid") long uid) {
Id<Switch, UUID> deviceId = Id.valueOf(UUID.fromString(uid));

SwitchService service = get(SwitchService.class);
Switch device = service.get(deviceId);
if (device == null) {
    throw new NotFoundException("device with id '" + id + "' not found");
}

service.delete(deviceId);

return Response.ok().build();
}

private ResponseBuilder response(Switch s, ObjectMapper mapper) {
    ObjectNode r = mapper.createObjectNode();
r.put("item", json(s, mapper()));
    return ok(r.toString());
}

static JsonNode json(Switch s, ObjectMapper mapper) {
    ObjectNode node = mapper.createObjectNode();
    node.put("uid", s.getId().getValue().toString());
    node.put("name", s.name());
}

211
Posting Alerts

In order to illustrate how alerts may be posted using the AlertService published by the controller, SwitchManager of the sample application will post an alert if a device is read with an. See Alert Logging on page 20 to get more information.

At this point SwitchManager already depends on the AlertService, so it is ready to use this service. The following listing illustrates an extract of a modified SwitchManager that posts an alert when a device is retrieved.

SwitchManager.java Posting Alerts:

```java
package com.hp.hm.impl;

import com.hp.sdn.adm.alert.AlertService;
import com.hp.sdn.adm.alert.AlertTopic;
...
public class SwitchManager implements SwitchService {
    ...
    private AlertService alertService;
    private AlertTopic alertTopic;
    ...
    public void setAlertService(AlertService alertService) {
        this.alertService = alertService;
        alertTopic = alertService.registerTopic("of_controller_hm", "health-monitor", "Alerts from the health monitor application");
    }
    ...
    @Override
    public Switch get(Id<Switch, UUID> id) {
        ...
        if (alertService != null) {
            String source = "OpenFlow Switch: " + id.getValue();
            String data = "Switch Retrieved!!";
            alertService.post(Severity.WARNING, alertTopic, source, data);
        }
    }
    ...
    }
```
When the optional `AlertService` is set, an alert topic is registered using the `AlertService`. This registration process will return the alert topic to use when the alert is posted. Alert topics are persistent, thus if the topic was already registered, registering it again will have no effect.

Since `AlertService` is optional in `SwitchManager`, the alert will be posted just if the service is available, thus a check for null is needed before posting the alert.

---

**NOTE**

As mentioned before, a better design would make use of the decorator pattern [XXX] to decorate business logic with optional dependencies so no check for null is needed and logics with different concerns are separated.

Optional services are bound/unbound in a multi-thread environment. An optional service may become unavailable at any time and thus synchronization methods (Avoided here for simple illustration purposes) need to be put in place.

To try the new alert feature use the Rsdoc to add and modify an OpenFlow switch so an alert is generated.

1. Build and install the application as described in Building the Application on page 146 and Installing the Application on page 147.
2. Open the HP VAN SDN Controller’s Rsdoc and authenticate as illustrated in Trying the REST API with RSdoc on page 196.
3. Add (POST) a device using the following JSON document: `{ "switch": {"name": "OpenFlow switch 1"}}`(as illustrated in Figure 60)

**Figure 60 Adding OpenFlow Switch**
4. Retrieve (GET) the device (as illustrated in Figure 61).

**Figure 61 Updating OpenFlow Switch**

![GET /health[/uid](https://i.imgur.com/7d901-8888-4ec7-856c-fb80b8b8b8b8)

**Implementation Notes**

Normal Response Code(s): ok (200)

Error Response Codes: badRequest (400), unauthorized (401), forbidden (403), badMethod (405), serviceUnavailable (503), itemNotFoundException (404)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uid</td>
<td>57e7d901-8888-4ec7-856c-fb80b8b8b8b8 (the Switch unique identifier)</td>
</tr>
</tbody>
</table>

**Request URL**

https://[SDN_CONTROLLER_ADDRESS]:8443/sdn/ui/app

**Response Body**

```json
{
    "uid": "57e7d901-8888-4ec7-856c-fb80b8b8b8b8",
    "name": "Switch 57e7d901-8888-4ec7-856c-fb80b8b8b8b8"
}
```

**Response Code**

200

5. Open the HP VAN SDN Controller’s alerts view:

https://[SDN_CONTROLLER_ADDRESS]:8443/sdn/ui/app (as illustrated in Figure 62)
Auditing with Logs

In order to illustrate how audit logs may be posted using the AuditLogService published by the controller, SwitchManager of the sample application will post an audit log when a device is added. See Audit Logging on page 19 to get more information.

The AuditLogService dependency must be added as any other service to consume; see Consuming Services with OSGi Declarative Services on page 166. The following listings illustrates an extract of a modified SwitchManager that posts audit logs. It assumes you’ve implemented SwitchComponent.java shown below.

SwitchManager.java Posting Audit Logs:

```java
package com.hp.hm.impl;

import com.hp.sdn.adm.auditlog.AuditLogService;
import org.apache.felix.scr.annotations.Reference;
import org.apache.felix.scr.annotations ReferenceCardinality;
import org.apache.felix.scr.annotations.ReferencePolicy;
...

public class SwitchManager implements SwitchService {
    ...
    private AuditLogService auditLogService;
    ...
    public void setAuditLogService(AuditLogService auditLogService) {
        this.auditLogService = auditLogService;
    }

    @Override
```
public Switch create(String name) {
    ...

    if (auditLogService != null) {
        // TODO: com.hp.sdn.rs.misc.ControllerResource (super class of
        // RESTful Web Services) offers a method to retrieve the
        // authenticated user: getAuthRecord(). SwitchService may be
        // modified to receive com.hp.api.auth.Authentication as
        // parameter and extract the authenticated user from there.
        String user = "hm";
        String source = "Health Monitor";
        String activity = "Device Added";
        String description = "OpenFlow Switch: " + deviceToAdd.getId().getValue();
        auditLogService.post(user, source, activity, description);
    }

    return deviceToAdd;
}

SwitchComponent.java Consuming AuditLogService:

package com.hp.hm.impl;

import com.hp.sdn.adm.auditlog.AuditLogService;
...
@Component
@Service
public class SwitchComponent implements SwitchService {
    ...
    @Reference(policy = ReferencePolicy.DYNAMIC,
    cardinality = ReferenceCardinality.OPTIONAL_UNARY)
    private volatile AuditLogService auditLogService;

    @Activate
    public void activate() {
        delegate = new SwitchManager(systemInformationService);
        delegate.setAlertService(alertService);
        delegate.setAuditLogService(auditLogService);
    }
    ...
    protected void bindAuditLogService(AuditLogService service) {
        auditLogService = service;
        if (delegate != null) {
            delegate.setAuditLogService(service);
        }
protected void unbindAuditLogService(AuditLogService service) {
    if (auditLogService == service) {
        auditLogService = null;
        if (delegate != null) {
            delegate.setAuditLogService(null);
        }
    }
}
...

To try the new audit log feature follow the same steps from Posting Alerts on page 212 to add an OpenFlow switch so an audit log is generated.

Figure 63 Audit Logs View

Debugging with Logs

The HP VAN SDN Controller uses the Simple Logging Façade for Java (SLF4J) [44] logging framework to generate support logs. No extra configuration is needed to enable an application to create loggers. The following listing shows an example.

SwitchManager.java Using Logging:

```java
package com.hp.hm.impl;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
...

public class SwitchManager implements SwitchService {
```
private Logger logger;

public SwitchManager() {
    ...
    // The LoggerFactory may be wrapped by a class in charge of providing
    // loggers to guarantee loggers are created in a consistent manner
    logger = LoggerFactory.getLogger(getClass());
}
...

@Override
public Switch add(Switch device) {
    ...
    logger.info("Device {} added", device);
    ...
}
...
...

Log entries are stored in the file logs/log.log; see the HP VAN SDN Controller Admin Guide [29] to get instructions about exporting support logs. If a secure shell (SSH) session is opened to the controller the log entries may found at /opt/sdn/virgo/serviceability/logs/log.log.

Using OpenFlow

The sample application was described as an application to monitor reachability status of OpenFlow switches. So far no monitor capabilities have been included. The OpenFlow Controller published by the HP VAN SDN Controller will be used to accomplish such monitoring. The following example is a hypothetical implementation not reflected in the generated sample application.

NOTE:

This may not be the best way to monitor reachability status and such monitoring may only be of concern to network management applications, however it represents a good example for interacting with the OpenFlow controller.

The OpenFlow Controller is responsible for accepting and maintaining connections from OpenFlow-capable devices, and providing basic services to SDN Applications. See High Availability

Role orchestration

Role Orchestration Service provides a federated mechanism to define the role of teamed controllers with respect to the network elements in the controlled domain. The role that a controller assumes in relation to a network element would determine whether it has abilities to write and modify the configurations on the network element, or has only read-only access to it.
As a preparation to exercise the Role Orchestration Service (ROS) in the HP VAN SDN Controller, there are two pre-requisite operations that needs to be carried out beforehand:

3) Create controller team: Using the teaming interfaces, a team of controllers need to be defined for leveraging High Availability features.
4) Create Region: the network devices for which the given controller has been identified as a master are grouped into “regions”. This grouping is defined in the HP VAN SDN Controller using the Region interface detailed in subsequent sections.

Once the region definition(s) are in place, the ROS would take care of ensuring that a master controller is always available to the respective network element(s) even when the configured master experiences a failure or there is effectively a disruption of the communication channel between the controller and the network device(s).

**Failover:** ROS would trigger the failover operation in two situations:

3) Controller failure: The ROS detects the failure of a controller in a team via notifications from the teaming subsystem. If the ROS determines that the failed controller instance was master to any region, it would immediately elect one of the backup (slave) controllers to assume the mastership over the affected region.
4) Device disconnect: The ROS instance in a controller would get notified of a communication failure with network device(s) via the Controller Service notifications. It would instantly federate with all ROS instances in the team to determine if the network device(s) in question are still connected to any of the backup (slave) controllers within the team. If that is the case, it would elect one of the slaves to assume mastership over the affected network device(s).

**Failback:** When the configured master recovers from a failure and joins the team again, or when the connection from the disconnected device(s) with the original master is resumed, ROS would initiate a failback operation i.e. the mastership is restored back to the configured master as defined in the region definition.

ROS exposes API’s through which interested applications can:

7) Create, delete or update a region definition
8) Determine the current master for a given device identified by a datapathId or IP address
9) Determine the slave(s) for a given device identified by a datapathId or IP address
10) Determine if the local controller is a master to a given device identified by a datapath
11) Determine the set of devices that a given controller is playing the master or slave role.
12) Register for region and role change notifications.

Details of the RegionService and RoleService APIs may be found at the Javadocs provided with the SDK. See Javadoc on page 9 for details.

**Illustrative usages of Role Service API’s**

- To determine the controller which is currently playing the role of Master to a given datapath, applications can use the following API’s depending on the specific need:

  ```java
  import com.hp.sdn.adm.role.RoleService;
  import com.hp.sdn.adm.system.SystemInformationService;
  ...
  public class SampleService {
      // Mandatory dependency.
  ```
private final SystemInformationService sysInfoService;

// Mandatory dependency.
private final RoleService roleService;

public void doAct() {
  IpAddress masterIp = roleService.getMaster(dpid).ip();
  if(masterIp.equals(sysInfoService.
    getSystem().getAddress())){
    log.debug("this controller is the master to {}", dpid);
    // now that we know this controller has master privileges
    // we could for example initiate write operations on the
    // datapath - like sending flow-mods
  }
}

- To determine the role that a controller is playing with respect to a given datapath

import com.hp.of.lib.msg.ControllerRole;
import com.hp.sdn.adm.role.RoleService;
import com.hp.sdn.region.ControllerNode;
import com.hp.sdn.region.ControllerNodeModel;
...
public class SampleService {
  // Mandatory dependency.
  private final RoleService roleService;
  public void doAct() {
    ...
    ControllerNode controller = new ControllerNodeModel("10.1.1.1");
    ControllerRole role = roleService.getCurrentRole(controller,deviceIp);
    switch(role){
      case MASTER:
        // the given controller has master privileges
        // we can trigger write-operations from that controller
        ...
        Break;
      Case SLAVE:
        // we have only read privileges
        ...
        break;
      default:
        // indicates the controller and device are not associated
// to any region.
break;
}
}

Notification on Region and Role changes

Applications can express interest in region change notifications using the addListener(...) API in RegionService and providing an implementation of the RegionListener. A sample listener implementation is illustrated in the following listing:

Region Listener Example:

```java
import com.hp.sdn.adm.region.RegionListener;
import com.hp.sdn.region.Region;
...
public class RegionListenerImpl implements RegionListener {
...
@Override
public void added(Region region) {
    log.debug("Master of new region: {}", region.master());
}

@Override
public void removed(Region region) {
    log.debug("Master of removed region: {}", region.master());
}
}
```

Similarly applications can express interest in role change notifications using the addListener(...) API in RoleService and providing an implementation of the RoleListener. A sample listener implementation is illustrated in the following listing:

Role Listener Example:

```java
import com.hp.sdn.adm.role.RoleEvent;
import com.hp.sdn.adm.role.RoleListener;
...
public class RoleListenerImpl implements RoleListener {
...
@Override
public void rolesAsserted(RoleEvent roleEvent) {
    log.debug("Previous master: {}", roleEvent.oldMaster());
    log.debug("New master: {}", roleEvent.newMaster());
    log.debug("Affected datapaths: {}", roleEvent.datapaths());
}
}
```

OpenFlow on page 23 for more details. The ControllerService API provides a common facade for consumers to interact with the OpenFlow Controller. Applications and Services register with the controller as specific types of listener:

- **DataPathListener**—To receive events about datapath connection.
• **MessageListener**—To receive events about OpenFlow messages received by the controller from connected datapaths.

• **SequencedPacketListener**—To participate in the processing of Packet-In messages

• **FlowListener**—To receive events about flow.

`DataPathListener` will be used to monitor connections and thus translate connected devices to reachable devices. Even though just `DataPathListener` will be shown here, using the other listeners is a matter of creating a variation of what is shown.

The following listings illustrate an extract of the implementation of `SwitchManager` and `SwitchComponent` consuming the `ControllerService` to monitor datapath connections.

**SwitchManager.java** Subscribing a `DataPathListener` to the `ControllerService`:

```java
package com.hp.hm.impl;

import com.hp.of.ctl.ControllerService;
import com.hp.of.ctl.DataPathEvent;
import com.hp.of.ctl.DataPathListener;
import com.hp.of.ctl.OpenflowEventType;
import com.hp.of.ctl.QueueEvent;
import com.hp.of.lib.dt.DataPathInfo;
...
public class SwitchManager implements SwitchService {
    ...
    private DataPathListener dataPathListener;

    public SwitchManager(SystemInformationService systemInformationService) {
        ...
        dataPathListener = new DataPathListenerImpl();
    }
    ...
    @Override
    public List<Switch> find(SwitchFilter filter,
        SortSpecification<SwitchSortKey> sortSpecification) {
        // In a real application a database may be used: filter would be
        // mapped to predicates and sortSpecification to sorting clauses.
        List<Switch> switches = new ArrayList<Switch>(devices.values());
        // At this point just the MAC Address filter is used so a temporal
        // implementation is also used (NOTE: This is not a proper way of
        // implementing filtering).
        // -----
        if (filter != null && filter.getMacAddressCondition() != null) {
            List<Switch> toDelete = new ArrayList<Switch>();
            MacAddress filterMacAddress = filter.getMacAddressCondition().
                .getValue();
```
EqualityCondition.Mode mode = filter.getMacAddressCondition().getMode();
for (Switch device : switches) {
    if (device.getMacAddress().equals(filterMacAddress)) {
        if (mode == EqualityCondition.Mode.UNEQUAL) {
            toDelete.add(device);
        }
    } else {
        if (mode == EqualityCondition.Mode.EQUAL) {
            toDelete.add(device);
        }
    }
}
switches.removeAll(toDelete);

// -----

return switches;

... private Switch getByMacAddress(MacAddress macAddress) {
    SwitchFilter filter = new SwitchFilter();
    filter.setMacAddressCondition(
        new EqualityCondition<MacAddress>(macAddress,
                                           EqualityCondition.Mode.EQUAL));
    List<Switch> switches = find(filter, null);
    if (!switches.isEmpty()) {
        return switches.get(0);
    }
    return null;
}

void startHandlingControllerEvents(ControllerService controllerService) {
    controllerService.addDataPathListener(dataPathListener);
    Set<DataPathInfo> dataPaths = controllerService.getAllDataPathInfo();
    Set<MacAddress> connectedSwitches = new HashSet<MacAddress>();
    for (DataPathInfo dataPathInfo : dataPaths) {
        connectedSwitches.add(dataPathInfo.dpid().getMacAddress());
    }
    for (Switch device : find(null, null)) {
        if (connectedSwitches.contains(device.getMacAddress())) {
            device.setActiveState(ActiveState.ON);
void stopHandlingControllerEvents(ControllerService controllerService) {
    controllerService.removeDataPathListener(dataPathListener);
}

private class DataPathListenerImpl implements DataPathListener {

    @Override
    public void queueEvent(QueueEvent event) {
    }

    @Override
    public void event(DataPathEvent event) {
        if (event.type() == OpenflowEventType.DATAPATH_CONNECTED ||
            event.type() == OpenflowEventType.DATAPATH_DISCONNECTED) {
            Switch device = etByMacAddress(event.dpid().getMacAddress());
            if (device != null) {
                if (event.type() == OpenflowEventType.DATAPATH_CONNECTED) {
                    device.setActiveState(ActiveState.ON);
                } else {
                    device.setActiveState(ActiveState.OFF);
                }
                update(device);
            }
        }
    }
}

SwitchComponent.java Consuming ControllerService:

package com.hp.hm.impl;

import com.hp.of.ctl.ControllerService;
...
@Component
@Service
public class SwitchComponent implements SwitchService {
    ...
    @Reference(policy = ReferencePolicy.DYNAMIC,
        cardinality = ReferenceCardinality.MANDATORY_UNARY)
private volatile ControllerService controllerService;
...

@Activate
public void activate() {
    delegate = new SwitchManager(systemInformationService);
    delegate.setAlertService(alertService);
    delegate.setAuditLogService(auditLogService);
    delegate.startHandlingControllerEvents(controllerService);
}

@Deactivate
public void deactivate() {
    delegate.stopHandlingControllerEvents(controllerService);
    delegate = null;
}
...

From the previous listings it can be seen that the MAC Address is used to relate connected devices to the devices managed by the sample application.

Some dependencies need to be resolved to use the OpenFlow controller services. Open the `hm-bli/pom.xml` file and add the XML extract from the following listing to the `<dependencies>` node. After updating the POM file update the Eclipse project dependencies (see Updating Project Dependencies on page 146).

OpenFlow Controller Dependencies:

```xml
<dependency>
    <groupId>com.hp.sdn</groupId>
    <artifactId>sdn-of-lib</artifactId>
    <version>${sdn.version}</version>
</dependency>
<dependency>
    <groupId>com.hp.sdn</groupId>
    <artifactId>sdn-of-ctl</artifactId>
    <version>${sdn.version}</version>
</dependency>
```

Real OpenFlow devices connected to the HP VAN SDN Controller are needed to try the monitoring capability added in this section. An alternative to real devices is using Mininet [45] (Used in this example) to create a realistic virtual network.

1. Follow the steps from Posting Alerts on page 212 to add an OpenFlow switch using the HP VAN SDN Controller’s Rsdoc. Just add the device, no need to modify it since the active state will be automatically updated based on the device connectivity status. Make sure the MAC Address used in the posted JSON document (illustrated in Figure 60) matches one of the devices that will be later connected to the controller.

2. Connect at least one OpenFlow-capable device to the HP VAN SDN Controller’s with the MAC Address used in previous step. Make sure the device is connected by checking the HP VAN SDN Controller’s topology view as illustrated in Figure 64. In this example two OpenFlow-
capable devices virtualized by Mininet [45] with MAC Addresses 00:00:00:00:00:01 and 00:00:00:00:00:02 were connected.

**Figure 64 OpenFlow Topology View**

![OpenFlow Topology View](image)

3. Verify the active state of the device added in Step 1 has been updated using the Rsdoc as illustrated in Figure 65.

**Figure 65 Sample Application OpenFlow Devices**

![Sample Application OpenFlow Devices](image)
After disconnecting the devices from the HP VAN SDN Controller (Stopping Mininet [45] in case of a virtualized network) the device’s active state should be updated to OFF.

**Application Manager Events/State**

In addition to the OSGI application service events, developers have access to SDN application events and SDN application state. For example, applications can query their own state during deactivation to perform pre-uninstall and/or pre-upgrade work.

The following is an example of how to listen to application events:

```java
private ApplicationService as;
private AppEventListener listener = new MyAppEventListener();

private class MyAppEventListener implements AppEventListener {
    @Override
    public void handleAppEvent(ApplicationEventType e, Application app) {
        if (app.id().equals("my-app-id")) {
            if (e == ApplicationEventType.UNINSTALLING) {
                // handle event
            } else if (e == ApplicationEventType.UPGRADING) {
                // handle event
            }
        }
    }
}

protected void bindAppService(ApplicationService as) {
    this.as = as;
    as.addApplicationListener(listener);
}

protected void unbindAppService(ApplicationService as) {
    this.as = null;
}

@Deactivate
protected void deactivate() {
    if (as != null)
        as.removeAppEventListener(listener);
}
```
The following is an example of how to query an application’s state:

```java
ApplicationService as;

protected void bindAppService(ApplicationService as) {
    this.as = as;
}

protected void unbindAppService(ApplicationService as) {
    this.as = null;
}

@Deactivate
protected void deactivate() {
    if (as != null) {
        Application.State myState = as.state("my-app_id");
        if (myState == State.UNINSTALLING) {
            // handle uninstalling
        } else if (myState == State.UPGRADING) {
            // handle upgrading
        }
    }
}
```
The following information describes how to test SDN applications by executing Unit Test and enabling remote debugging in the controller.

**Unit Testing**

Unit tests are automatically run when building the application; see Building the Application on page 146. There is a version of this command to avoid running unit tests:

**Building Application Ignoring Unit Test:**

$$\text{mvn clean install -Dmaven.test.skip=true}$$

The Building Application Ignoring Unit Test command is not recommended but it could be useful in cases where the unit test is temporarily broken.

Unit test is part of the project’s test directory which is configured as a source file in Eclipse. Just by following the Maven directory structure conventions, when Maven generates the Eclipse projects it configures the test folder as a source folder. Thus, to run unit tests within Eclipse right click on a specific project and then select *Run As → JUnit Test* as illustrated in Figure 66 and Figure 67.

**Figure 66 Running Unit Test within Eclipse (Step 1)**
There are several tools that calculate unit test coverage which are very useful. EclEmma [46] is a free Java code coverage tool for Eclipse, available under the Eclipse Public License. It brings code coverage analysis directly into the Eclipse workbench. When EclEmma is installed as an Eclipse plug-in, the unit test needs to be rerun using EclEmma as illustrated in Figure 68 and Figure 69. See Installing Eclipse Plug-ins on page 246 to follow instructions about installing a plug-in; use http://update.eclemma.org as the repository location.
Figure 68 Unit Test Coverage

Figure 69 Unit Test Coverage Result
Remote Debugging with Eclipse

It is possible to enable remote debugging with the controller; to do so setup a debugging session with the controller: Go to Run → Debug Configurations… to open the debug configurations dialog and select Remote Java Applications, click New as illustrated in Figure 70.

Figure 70 Remote Java Application’s Debug Configuration

Set the SDN Controller configuration with the data shown in Figure 71. Set any name for the configuration and use the IP Address of the controller as the configuration host. Then click Apply. A new configuration is displayed, with the name previously set, being added under Remote Java Application as illustrated in Figure 72. Click Debug to start. From now on, every time to remotely debug the controller, open the Debug Configurations dialog, select the configuration just created (HpSdnController) and execute Debug.
Figure 71 HP VAN SDN Controller’s Remote Debug Configuration

Figure 72 HP VAN SDN Controller Saved Remote Debug Configuration
Now add a break point and verify that the controller stops at that point. You may skip the rest of this information if familiar with Eclipse’s debug perspective. Use the application developed on page 126 at the point of section GUI-Specific REST API on page 204. The reason to do so is because at that point, the application generates a very simple user interface with a single button that displays a message retrieved from the server via RESTful web services; and this adds a breakpoint in that REST API. You may not be able to follow the remaining of this section using such application in that particular state, however you can follow the section just by adding a break point in any code that is executed in the controller (Which is any Java code in your application); you just need to figure out an action that triggers the code you want to remotely debug.

Open a Java file and add a break point. Following the sample application we will open the REST API used by the GUI, SwitchResource from module hm-ui, and add a break point in the only RESTful method there as illustrated in Figure 73.

Figure 73 Adding Break Point to SwitchViewResource.java

Now open the application and click on the Refresh DeviceHealth button that displays a message retrieved from the REST API where the break point was just added. Figure 74 shows the sample application’s view. After clicking Refresh DeviceHealth, notice that a confirmation message from Eclipse requesting to change to the debug perspective, Figure 75. That means the controller hit the break point and now it can run step by step. Select Yes to continue.
The code stopped at the break point can be seen, as in Figure 76, however, as we can see in Figure 75 the source file was not found. If the source code cannot be seen add the Eclipse projects as Source Lookup Path: See Attaching Source Files when Debugging on page 248.
Figure 77 shows the code stopped at the break point and the state of the SDN Controller’s view that depends on the code being debugged. It is only until we resume execution by clicking the Resume tool bar action that the controller’s view completes as illustrated in Figure 78.
Figure 77 Controller’s View Waiting for Code Being Debugged

![Image of controller's view waiting for code being debugged](image1)

Figure 78 SDN Controller’s View Completed after Execution Resumed

![Image of SDN controller's view completed after execution resumed](image2)
8 Built-In Applications

The HP VAN SDN Controller ships with a default set of core network service components, which provide an out-of-box experience in terms of enabling connectivity across network applications in the Openflow network. The details of each are captured below.

Node Manager

Node network service component is responsible for creating and maintaining the node table. Each end-host (called a node) is uniquely identified by the combination of IP address and network segment. The data stored for each node includes the node’s MAC address, network interface, timeout value, and the current location.

The node table sample data as shown in Table 10.

Table 10 Node Table

<table>
<thead>
<tr>
<th>IP Address</th>
<th>MAC</th>
<th>Segment ID</th>
<th>Device ID</th>
<th>Interface</th>
<th>Timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.250.100.1</td>
<td>00:af:cd:12:10:01</td>
<td>100</td>
<td>00:ae:c7:de:02:01:02:03</td>
<td>3</td>
<td>300</td>
</tr>
<tr>
<td>10.250.100.2</td>
<td>00:af:cd:12:10:20</td>
<td>110</td>
<td>00:ae:c7:de:02:01:02:03</td>
<td>4</td>
<td>1200</td>
</tr>
</tbody>
</table>

Node Manager publishes the com.hp.sdn.node.NodeService API via OSGI declarative services and REST API. These APIs allow callers to perform queries against the node table.

Node Manager also publishes the com.hp.sdn.supplier.NodeSuppliersBroker, which allows callers to register themselves as a supplier of node-related information, using an instance of com.hp.sdn.node.NodeSupplierService. See Javadoc on page 9 for details.

Node Manager performs aging for all entries in the node table. If a node has not been updated for a period of time exceeding its timeout value, then that node will be removed from the table. This aging is performed to keep an accurate view of the live nodes on the controlled network.

OpenFlow Node Discovery

OpenFlow Node Discovery pushes flow-mods to controlled devices and listens for PACKET_IN messages in order to discover nodes on the controlled network. When hybrid.mode=true in the ControllerManager configuration, OpenFlow Node Discovery will push flows to controlled devices which send copies of the following packets to the controller:

- all ARP packets
- all DHCP packets from the DHCP server to end-hosts

OpenFlow Node Discovery listens for PACKET_IN messages which contain the ARP or DHCP protocol. If learn.ip=true in the OfIpDiscoveryComponent configuration, then OpenFlow Node Discovery will also listen for PACKET_IN messages which contain the IP protocol. No flows are explicitly pushed to controlled devices which copy all IP traffic to the controller, as that would drastically reduce network performance by overwhelming the control plane. When the
ControllerManager configuration has `hybrid.mode=false`, all packets are implicitly stolen to the controller and processed by OpenFlow Node Discovery.

Based upon the information supplied by these copied ARP, DHCP, and IP packets the OpenFlow Node Discovery application will register as a node supplier and supply updates to the node table. The timeout value for nodes discovered by each protocol is configurable, as shown in Table 11.

### Table 11 Node Timeout

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Timeout</th>
<th>Configuration</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARP</td>
<td>arp.age</td>
<td>OfArpDiscoveryComponent</td>
<td>5 minutes</td>
</tr>
<tr>
<td>DHCP</td>
<td>dhcp.age</td>
<td>OfDhcpDiscoveryComponent</td>
<td>24 hours</td>
</tr>
<tr>
<td>IP</td>
<td>lp.age</td>
<td>OfIpDiscoveryComponent</td>
<td>20 minutes</td>
</tr>
</tbody>
</table>

Node Manager will not update the node table for every PACKET_IN message it receives. Specifically, PACKET_IN messages are ignored if the connected port is identified as being part of the infrastructure by the Topology module.

Note that since these PACKET_IN messages represent copies of packets which have already been forwarded by the controlled device, no corresponding PACKET_OUT will be sent back to the device which sent the PACKET_IN.

### Link Manager

The Link network service component is responsible for creating and maintaining the infrastructure link table. Each infrastructure link is uniquely identified by source connection point and destination connection point. The data stored for each link includes the link type, which may be direct, multi-hop, or tunnel. A direct link represents a non-switched connection between two controlled devices. A multi-hop link represents a switched connection between two controlled devices, where the connection spans one or more uncontrolled switches. A tunnel link represents a configured tunnel between two devices.

The Link table sample data is shown in Table 12.

### Table 12 Link Table

<table>
<thead>
<tr>
<th>Source Device</th>
<th>Source Port</th>
<th>Destination Device</th>
<th>Destination Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>03:e7:00:26:f1:29:af:00</td>
<td>1</td>
<td>03:e7:00:23:47:ba:05:40</td>
<td>23</td>
</tr>
<tr>
<td>03:e8:00:23:47:ba:05:40</td>
<td>11</td>
<td>03:e8:00:26:f1:29:af:00</td>
<td>8</td>
</tr>
</tbody>
</table>

Link Manager publishes the `com.hp.sdn.link.LinkService` API via OSGI declarative services and REST API. These APIs allow callers to perform queries against the link table.

Link Manager also publishes the `com.hp.sdn.supplier.LinkSuppliersBroker`, which allows callers to register themselves as a supplier of link-related information, using an instance of `com.hp.sdn.link.LinkSupplierService`. See Javadoc on page 9 for details.
OpenFlow Link Discovery

OpenFlow Link Discovery pushes flow-mods to controlled devices and listens for PACKET_IN messages in order to discover links on the controlled network. When hybrid.mode=true in the ControllerManager configuration, OpenFlow Link Discovery will push a flow to controlled devices which steals all controller-generated link discovery packets to the controller. The controller-generated link discovery packets use a non-standard protocol (BDDP), which utilizes a payload format similar to LLDP. All discovery packets generated by the controller will be sent to either a link-local MAC address (to discover direct links) or a multicast MAC address (to discover multi-hop links). The multicast MAC address used for link discovery is 01:1B:78:E9:7B:CD. When the controller injects a discovery packet, the packet content contains the device ID which introduces the packet to the controlled network.

OpenFlow Link Discovery listens for PACKET_IN messages which contain the BDDP protocol. Each discovery packet has the source device ID embedded within its payload, and the destination device can be derived from the PACKET_IN message. This allows the OpenFlow Link Discovery application to populate the link table with information it learns from such received packets. Note that since these PACKET_IN messages are for controller-generated link discovery packets, no corresponding PACKET_OUT will be sent back to the device which sent the PACKET_IN.

OpenFlow Link Discovery periodically injects discovery packets into the controlled network to refresh the contents of the link table. Any links which are not refreshed at this periodic interval are considered to be invalid and are removed from the link table. Additionally, network events such as a port going down or a device going offline will cause relevant links to be removed.

Topology Manager

Topology Manager provides topology information of the control domain. It also facilitates shortest path traversals through the control domain by way of computing low cost next hops between any two elements in the control domain. Topology Manager computes the clusters and broadcast tree to avoid loops and broadcast storms.

- Provides a list of discovered ports on a given switch.
- Indicates whether a switch port is an edge port (connection point) or part of a link.
- Indicate whether a port is in a blocked or open state by determining whether ingress broadcast traffic is allowed through the port.
- Verifies if a path exists between two nodes.
- Identifies the shortest path between two nodes.
- Provides enumeration of the grouping of switches into clusters of strongly connected nodes.
- For a given switch provides cluster details it belongs.

Topology manager provides notifications to subscribed applications on changes in its broadcast tree and cluster, intelligent applications can be developed which takes proactive measures by way of subscribing for these topology re-computed notifications.

Services published by Topology Service

- Given two switches (s1, s2) this can indicate if they can be reached via directly connected paths.
• For a given switch \( s_1 \), can provide the list of ports that this service has discovered
• For two switches \( s_1, s_2 \), the service can indicate if they are "strongly connected" i.e. form part of same cluster.
• For a given \( \{\text{switch}, \text{port}\} \) pair, it would indicate if they participate as a "connection point" (if they form an 'edge port')
• For a given \( \{\text{switch}, \text{port}\} \) pair, this service can indicate if ingress broadcast is allowed through 'port'
  o Example: if one needs to flood out packets through a port, it can do a check using this API to see if broadcast would be possible through this port. If this API indicates negative, then it would mean the port is in blocked state.
• Provide hooks for interested components to get notified of topology changes

See Javadoc on page 9 for details of the API’s provided by Topology Manager.

Path Diagnostics

Path Diagnostics is a default end-user application using PathDiagnosticService API. Path Diagnostics determines and verifies the path taken by trace packets from a source host to a destination host. The application finds an existing flow that matches the description of the trace packet, clone it with higher priority and add an additional action to instruct the selected switch to send this packet back to the controller for status tally.

For REST command line API please refer to HP VAN SDN Controller Administrator Guide [9.6]

Path Daemon

Path Daemon is a path-paving application which listens for all ARP and IP PACKET_IN messages and attempts to push flow-mods to switches along the forwarding path to ensure that such packets get forwarded at line-rate. Path Daemon operates most optimally when the entire network is controlled by the controller team (i.e.: no uncontrolled switches) and the switches are interconnected at layer 2. Each PACKET_IN message processed by Path Daemon will result in a PACKET_OUT message and possibly a flow-mod getting pushed to one or more controlled devices.

By default, the Path Daemon application will push flow-mods that attempt to forward traffic using only MAC addressing and port. These flow-mods are only pushed when the ControllerManager configuration has hybrid.mode=false. Specifically, the flow-mods will match all packets that enter a specific switch on a specific port and they will match only packets with the source MAC address and destination MAC address from the PACKET_IN. Optionally, the PathDaemon configuration allows the source IP and destination IP fields to be included as matching criteria (when layer3.forward=true).

Any packets which match the flow-mod will be forwarded to a destination port determined by Path Daemon to get the packet to most optimally reach its intended destination. Each flow-mod is assigned an idle timeout value, which specifies how long the flow-mod will remain in the device if the flow-mod is not actively being used. Each flow-mod is also assigned a hard timeout value, which specifies how long the flow-mod will remain in the device (regardless of usage). The
PathDaemon configuration allows configuration of each of these parameters as `idle.timeout` (default 60 seconds) and `hard.timeout` (default 0, which implies infinite timeout).
Appendix A

Using the Eclipse Application Environment

This appendix describes some of the Eclipse [47] features that an SDN Controller Application developer will often use.

Importing Java Projects

To import an entire Eclipse project from an archive file, follow these steps:

1. Go to File → Import. The following dialog appears.

   Figure 79 Eclipse Source Selection Dialog (Import Java Project)

2. Select Existing Projects into Workspace. Then Click the button next.
3. Click *Browse* button and find the root folder (SDN Controller Application Workspace folder) on your hard disk. Several projects can be imported together depending on the selected root directory. Then click *OK* to select it.

4. Click *Finish* to perform the import.
Figure 82 Import Dialog (Import java Project)

Figure 83 Eclipse Imported Projects
Setting M2_REPO Classpath Variable

Go to Window → Preferences. Then add the location of Maven repository as illustrated in Figure 84.

![Setting M2_REPO Classpath Variable](image)

Installing Eclipse Plug-ins

Most plug-ins will have an update site, making it easy to add and update plug-ins within Eclipse.

1. Find the URL of the update site for the plug-in.
2. Go to Help → Install New Software… and create a connection to an update site within Eclipse by adding a repository, as in Figure 85. Use the URL from step 1 as the location.
3. Select the checkbox of the plug-in and follow the installation wizard.

Figure 86 Eclipse’s Plug-in Installation Wizard
Eclipse Perspectives

A perspective defines the initial set and layout of views in the Workbench window [47]. Within the window, each perspective shares the same set of editors. Each perspective provides a set of functionality aimed at accomplishing a specific type of task or works with specific types of resources. For example, the Java perspective combines views that you would commonly use while editing Java source files, while the Debug perspective contains the views used while debugging Java programs. Switching perspectives frequently while working in the Workbench is expected.

Perspectives control what appears in certain menus and toolbars. They define visible action sets, which can be changed to customize a perspective. A perspective that you build in this manner can be saved, making a custom perspective that can be opened again later.

Use Window → Open Perspective to open a perspective. Once a perspective is opened it is be placed in the tool bar to switch perspectives. See Figure 87.

Figure 87 Perspectives Tool Bar

Attaching Source Files when Debugging

When you are debugging a program if Eclipse doesn’t find the source files it will show something like Figure 88 (For example when debugging a remote program that was not started by Eclipse). To fix this:

1. Click the Edit Source Lookup Path… button from Figure 88 to open the Edit Source Lookup Path dialog.
2. Click Add button from the Edit Source Lookup Path dialog.

3. Select Java Project as the source.
4. Select projects.

**Figure 91 Lookup Path Resource Selections**

5. Confirm configuration.

**Figure 92 Source Lookup Path Confirmation**
Appendix B

Troubleshooting

Maven Cannot Download Required Libraries

Problem

This problem occurs when Internet access requires a proxy. Maven is unable to download required libraries due connection time outs.

Figure 93 Maven Problem: No Proxy Configured

The output shown in Figure 94 is also related to the proxy problem and it happens when Maven proxy configuration is incorrect.

Figure 94 Maven Problem: Invalid Proxy Configuration

Solution

Make sure the proper proxy is configured in Maven. To configure a proxy add the following Maven Proxy Configuration listing (with the proper information) to Maven settings.xml file located at maven installation directory (/etc/maven for Linux installations). Note <proxies> xml node is already in the file, so look for it and add the <proxy> node from the following listing.
Maven Proxy Configuration:

```xml
<proxies>
  <proxy>
    <id>optional</id>
    <active>true</active>
    <protocol>http</protocol>
    <username>proxyuser</username>
    <password>proxypass</password>
    <host>web-proxy.rose.hp.com</host>
    <port>8088</port>
    <nonProxyHosts>local.net|some.host.com</nonProxyHosts>
  </proxy>
</proxies>
```

Path Errors in Eclipse Projects after Importing

Problem

This problem occurs when the M2_REPO variable is not set in Eclipse.

Figure 95 Eclipse Missing M2_REPO Configuration Problem
Solution

SDN Controller Applications relies on Maven to resolve project dependencies, thus the Maven repository location must be configured in Eclipse. For more information see Setting M2_REPO Classpath Variable on page 246.
Bibliography


[34] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns Elements of Reusable Object-Oriented Software, Addison Wesley, 2007.


