HP VAN SDN Controller Programming Guide

Abstract

The HP VAN SDN Controller Appliance serves as a delivery vehicle for SDN solutions providing a platform for developing various flavors of network controllers, e.g. data-center, public cloud, private cloud, campus edge networks, etc. This document provides detailed documentation for writing applications to run on the HP VAN SDN Controller platform.

Part number: 5998-4920

Software version: 2.0.0

Document version: 3

Contents

1 Introduction

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

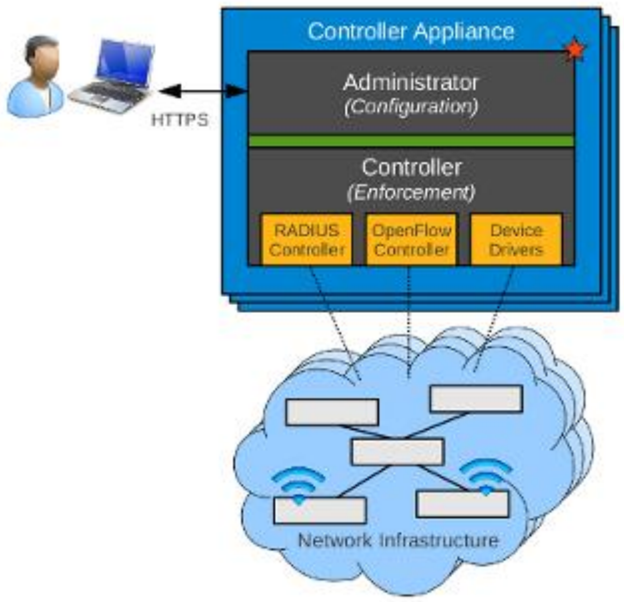
The base SDN Controller Appliance will serve as a delivery vehicle for SDN solutions. It aims to provide a platform for developing various flavours of network controllers, e.g. data-centre, public cloud, private cloud, campus edge networks, etc. This includes being an open platform for development of experimental and special-purpose network control protocols using a built-in OpenFlow controller.

The SDN Controller Appliance will meet certain minimum scalability requirements and it will provide ability to achieve higher scaling and high-availability requirements via a scale-out teaming model. In this model, the same set of policies will be 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 will consist of two major tiers. The upper Administrator tier will host 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, will host 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 appliance. Also, they are governed by a rule that no enforcement-related synchronous interaction will cross from the Controller to Administrator tier.

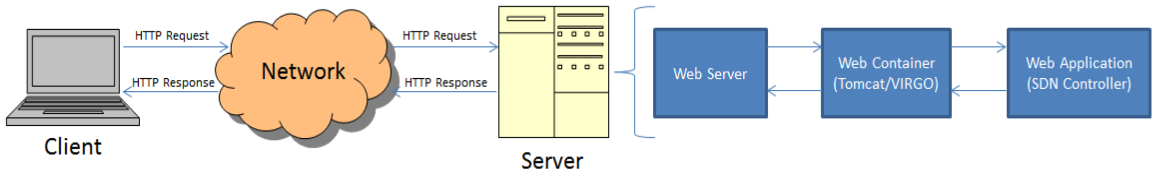
Figure 1 Controller Tiers



The Administration tier of the controller appliance will host a web-layer through which software modules installed on the appliance can expose REST API [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 Appliance.

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.

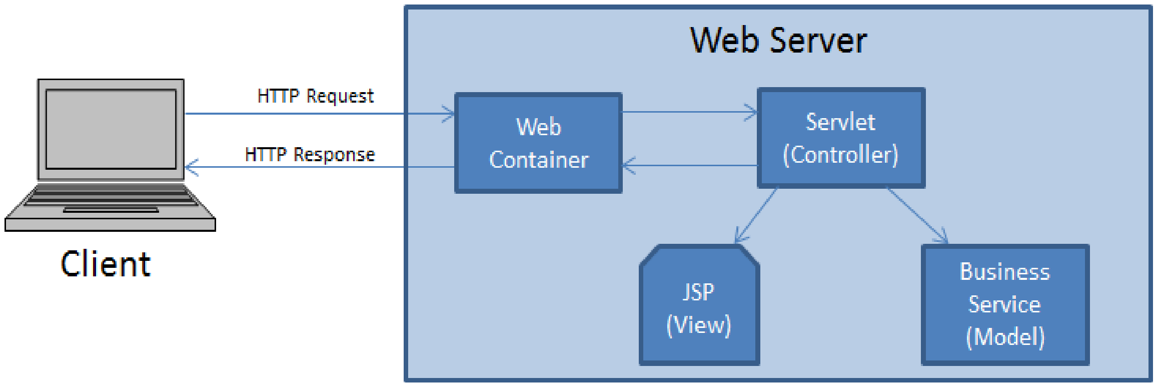
Figure 2 Web Application Architecture



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 don't 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.

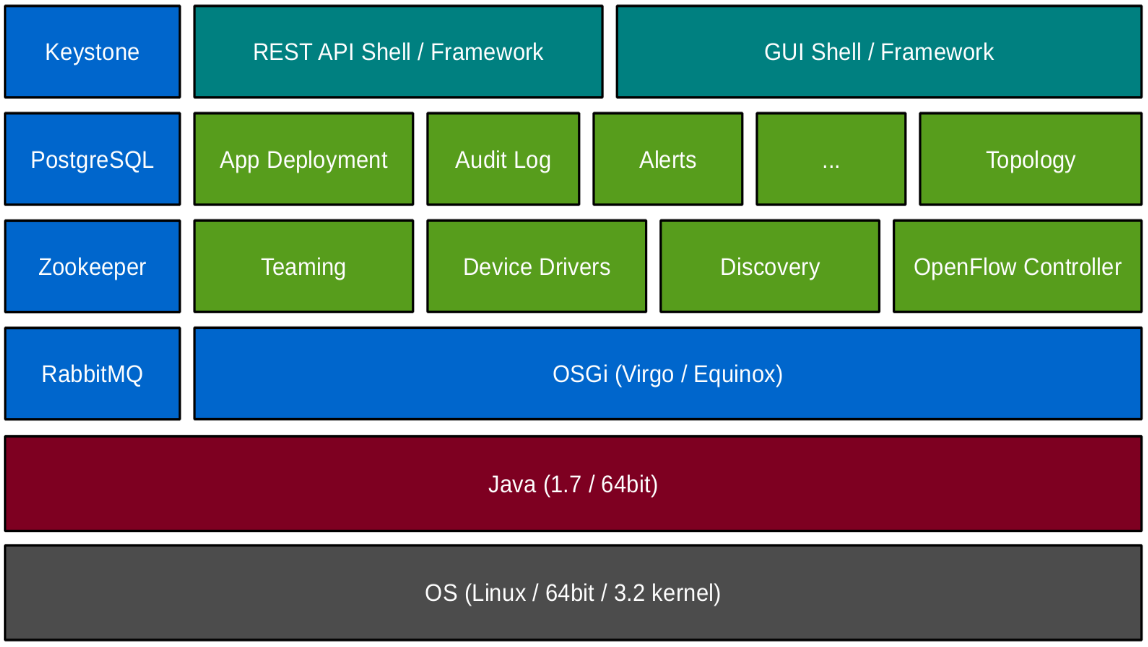
Figure 3 Web Application Model View Controller Pattern



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 may 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 may 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.

Figure 4 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 it is 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.

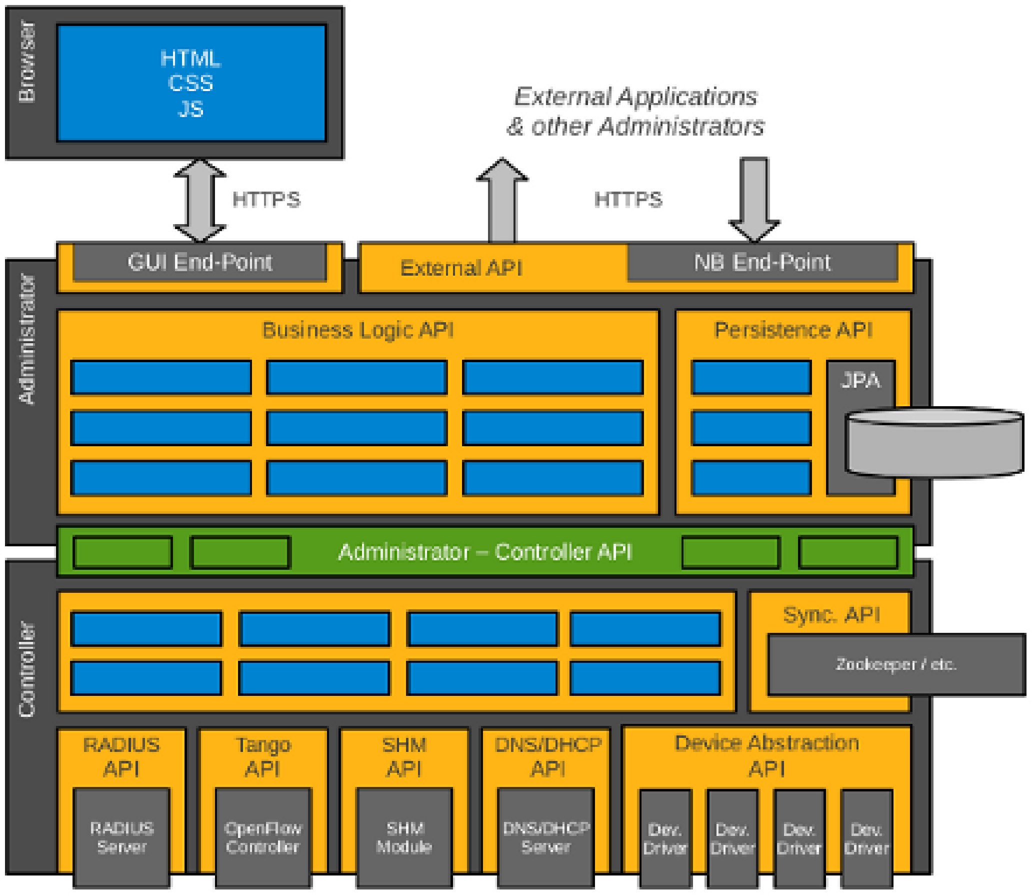
Keystone [9] external service provides authentication and high level authorization services. It supports token-based authentication scheme which is used to secure the RESTful web services (Or REST APIs) and the web user interfaces.

ZooKeeper [10] is a coordination service for distributed applications. It is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.

Apache Cassandra [11] 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 digested and broken down by the upper tier into something akin 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 [12] [13] 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 [12] [13], MSM controller software, and so on.

Figure 5 HP VAN SDN Controller Tiers



Internal Applications vs. External Applications

Internal applications ("Native" Applications / Modules) are deal to exert relatively fine-grained, frequent and low-latency control interactions with the environment, for example, handling packet-in events. Some key point 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 & HA 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.

External applications are suitable to exert relatively coarse-grained, infrequent and high-latency control interactions with the environment, for instance, path provisioning and flow inspections. Some key point to consider when developing external applications:

- Authored in any language capable of stablishing a secure HTTP connection. Example: Java, C, C++, Python, Ruby, C#, bash, and so on.
- Deployed on a platform of choice outside of the SDN Controller platform.
- Built atop REST API services exported and advertised by the platform and by other applications.
- Do not extend Java API ecosystem nor REST API and GUI surface of the SDN Controller.

External applications are deployed outside the HP VAN SDN Controller and they interact with it by consuming the RESTful Web Services published by the controller.

This guide will focus on the development of internal applications. For external applications consult the RESTful API documentation (or Rsdoc) as described at Rsdoc Live Reference on page 16.

2 Setting Environment

The suggested development environment is split into two: Test Environment and Development Environment. And it is recommended to use different machines. 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 [14] 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 wil 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) may 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 [15].

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 [15].

Development Environment

Pre-requisites

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

Operating System

Supported operating systems include:

- Windows 7or 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 [16] 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 [17].

To install Maven go to [17] 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 216.

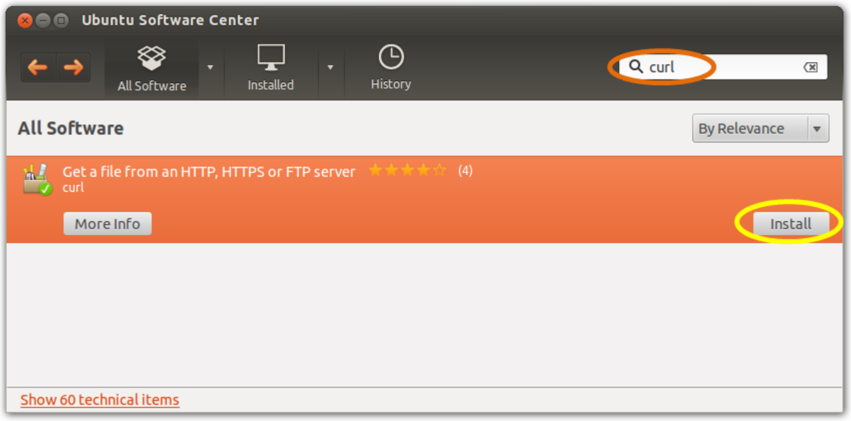
Maven 3.0.4 or newer is needed. To verify the installed version of Maven execute the following command:

```
$ mvn -versionCurl
```

Curl

Curl (or cURL) is a command line tool for transferring data with URL syntax. This tool is optional. Follow the instruction from [18] 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



HP VAN SDN Controller SDK

Download the HP VAN SDN Controller SDK from [19]. 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 be already installed and properly configured):

```
$ bin/install-sdk
```

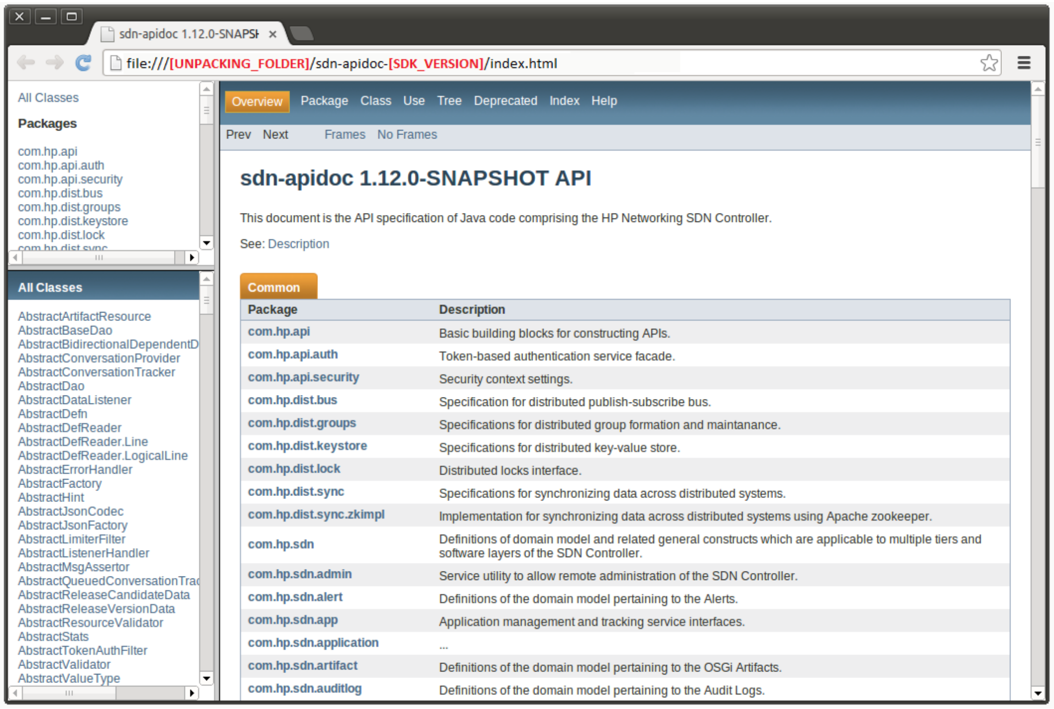
To verify that the SDK has been properly installed look for the HP SDN libraries installed in the local Maven repository at:

```
~/.m2/repository/com/hp.
```

Javadoc

Download HP VAN SDN Controller SDK API Documentation from [19]. The documentation is contained in the *hp-sdn-apidoc-*.jar* file (For example: *hp-sdn-apidoc-2.0.0.jar*). Unzip its contents in any location and open the *index.html* file. Figure 7 illustrates an example of the HP VAN SDN Controller documentation.

Figure 7 HP VAN SDN Controller Javadoc



sdn-apidoc 1.12.0-SNAPSH

file:///[UNPACKING_FOLDER]/sdn-apidoc-[SDK_VERSION]/index.html

All Classes

Packages

com.hp.api
com.hp.api.auth
com.hp.api.security
com.hp.dist.bus
com.hp.dist.groups
com.hp.dist.keystore
com.hp.dist.lock
com.hp.dist.sync

All Classes

AbstractArtifactResource
AbstractBaseDao
AbstractBidirectionalDependentD
AbstractConversationProvider
AbstractConversationTracker
AbstractDao
AbstractDataListener
AbstractDefn
AbstractDefReader
AbstractDefReader.Line
AbstractDefReader.LogicalLine
AbstractErrorHandler
AbstractFactory
AbstractHint
AbstractJsonCodec
AbstractJsonFactory
AbstractLimiterFilter
AbstractListenerHandler
AbstractMsgAssertor
AbstractQueuedConversationTrac
AbstractReleaseCandidateData
AbstractReleaseVersionData
AbstractResourceValidator
AbstractStats
AbstractTokenAuthFilter
AbstractValidator
AbstractValueType

Overview Package Class Use Tree Deprecated Index Help

Prev Next Frames No Frames

sdn-apidoc 1.12.0-SNAPSHOT API

This document is the API specification of Java code comprising the HP Networking SDN Controller.

See: Description

Common

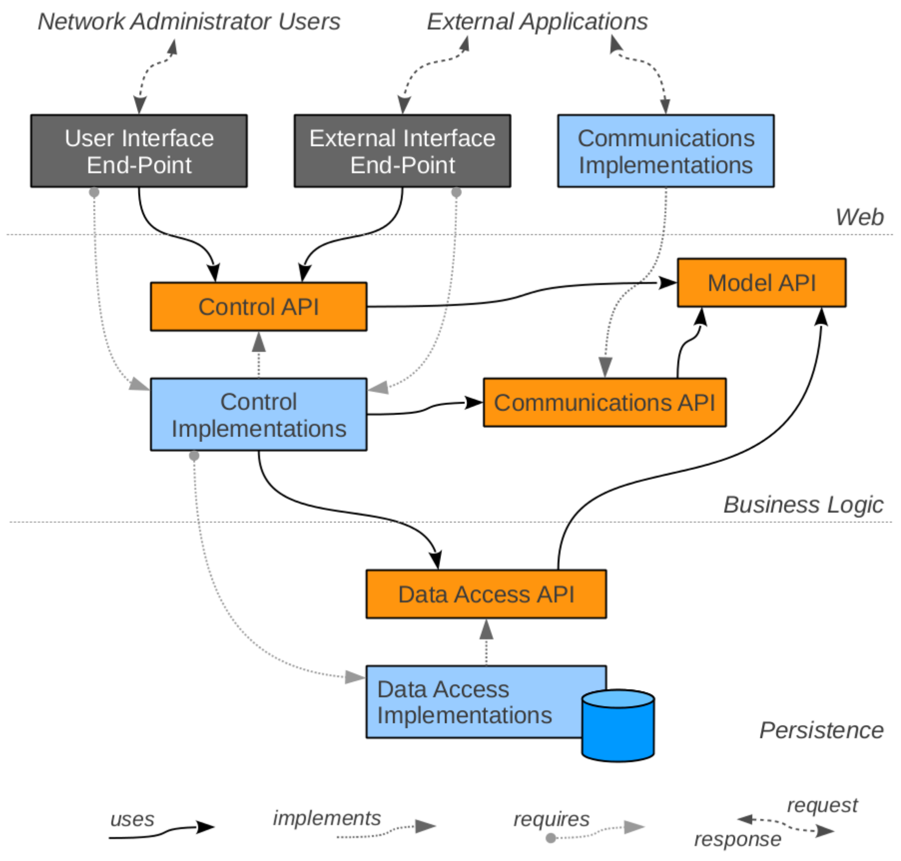
Package	Description
com.hp.api	Basic building blocks for constructing APIs.
com.hp.api.auth	Token-based authentication service facade.
com.hp.api.security	Security context settings.
com.hp.dist.bus	Specification for distributed publish-subscribe bus.
com.hp.dist.groups	Specifications for distributed group formation and maintanance.
com.hp.dist.keystore	Specifications for distributed key-value store.
com.hp.dist.lock	Distributed locks interface.
com.hp.dist.sync	Specifications for synchronizing data across distributed systems.
com.hp.dist.sync.zkimpl	Implementation for synchronizing data across distributed systems using Apache zookeeper.
com.hp.sdn	Definitions of domain model and related general constructs which are applicable to multiple tiers and software layers of the SDN Controller.
com.hp.sdn.admin	Service utility to allow remote administration of the SDN Controller.
com.hp.sdn.alert	Definitions of the domain model pertaining to the Alerts.
com.hp.sdn.app	Application management and tracking service interfaces.
com.hp.sdn.application	...
com.hp.sdn.artifact	Definitions of the domain model pertaining to the OSGi Artifacts.
com.hp.sdn.auditlog	Definitions of the domain model pertaining to the Audit Logs.

3 Developing Applications

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



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 comprise of interfaces and passive POJOs [20], 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.

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

HP VAN SDN Controller’s REST APIs are secured via a token-based authentication scheme. Openstack Keystone [9] is used to provide the token-based authentication.

This security mechanism:

- Provide user authentication functionality with RBAC support.
- Completely isolate security mechanism from the underlying RESTful API.
- Work well with Openstack Keystone (even though Keystone is not a requirement).
- Expose a RESTful API to allow any authentication server that implements this RESTful API to be hosted else where (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.

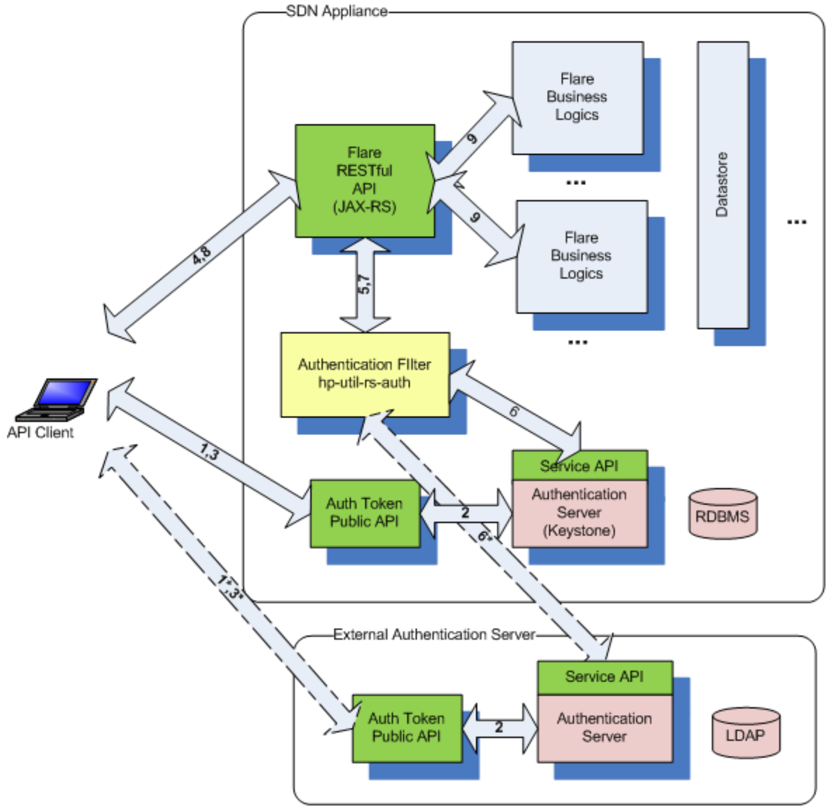
To achieve isolation of security aspects from the API, authentication information is encapsulated by a token that a user receives by presenting his/her credentials to an Authentication Server. The user then uses this token (via header X-Auth-Token) in any API call that requires authentication. The token 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 else where 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.

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

REST API

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 [21].

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 [21].

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.

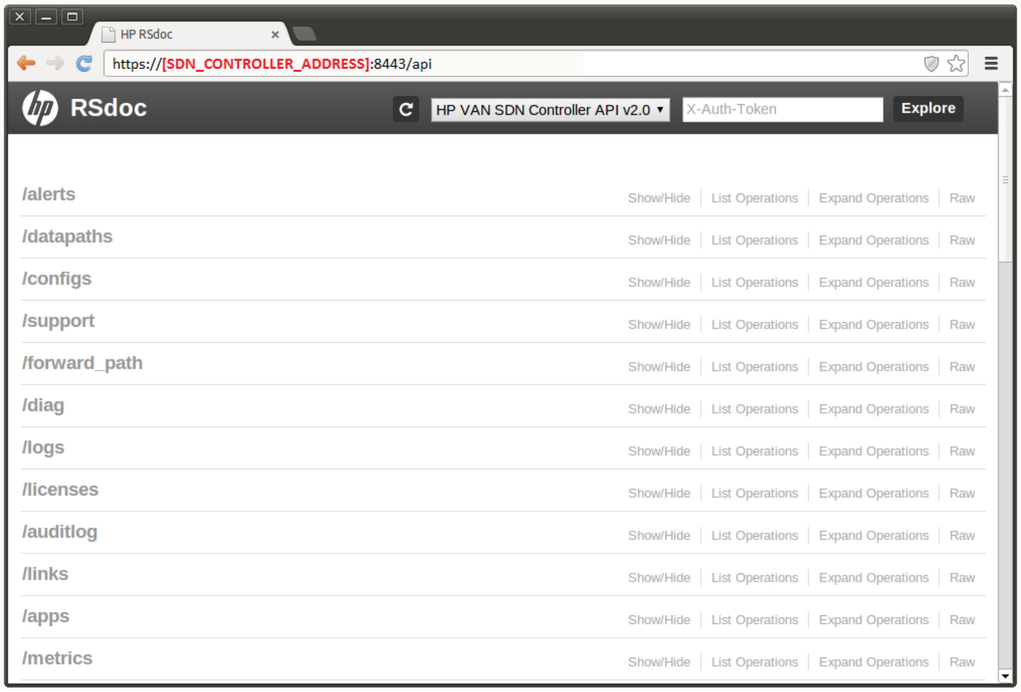
See [1] for guidelines to design REST APIs or RESTful Web Services and Creating REST API on page 137 for an example.

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 16.

Rsdoc

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

Figure 10 RSdoc



HP RSdoc

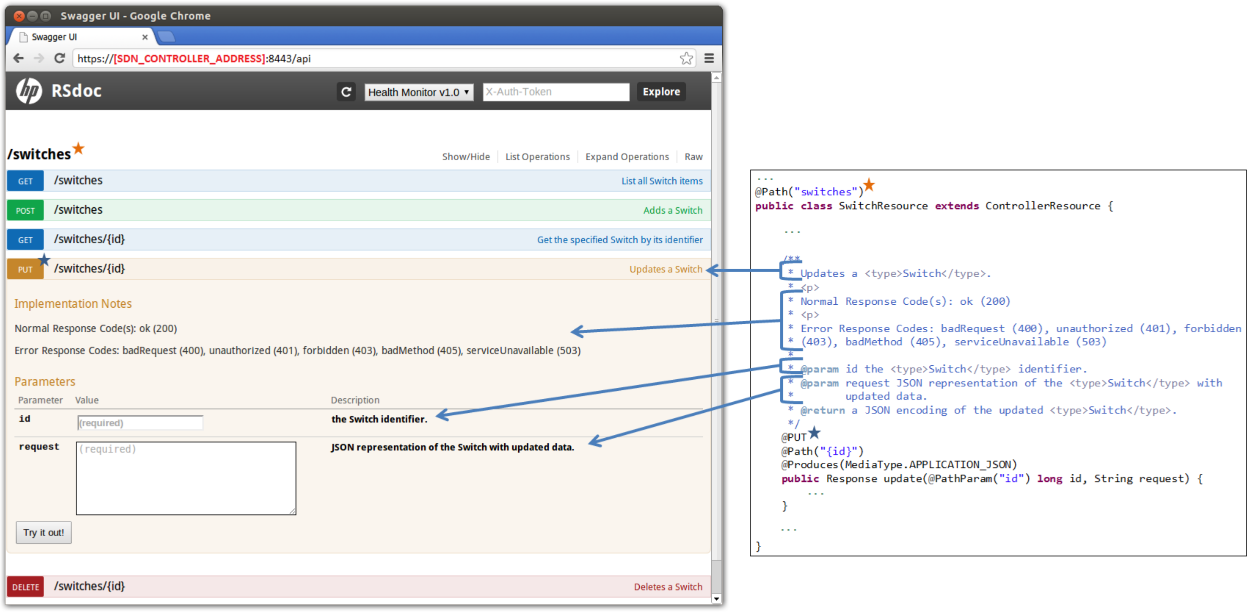
https://[SDN_CONTROLLER_ADDRESS]:8443/api

RSdoc — HP VAN SDN Controller API v2.0 — X-Auth-Token — Explore

/alerts	Show/Hide	List Operations	Expand Operations	Raw
/datapaths	Show/Hide	List Operations	Expand Operations	Raw
/configs	Show/Hide	List Operations	Expand Operations	Raw
/support	Show/Hide	List Operations	Expand Operations	Raw
/forward_path	Show/Hide	List Operations	Expand Operations	Raw
/diag	Show/Hide	List Operations	Expand Operations	Raw
/logs	Show/Hide	List Operations	Expand Operations	Raw
/licenses	Show/Hide	List Operations	Expand Operations	Raw
/auditlog	Show/Hide	List Operations	Expand Operations	Raw
/links	Show/Hide	List Operations	Expand Operations	Raw
/apps	Show/Hide	List Operations	Expand Operations	Raw
/metrics	Show/Hide	List Operations	Expand Operations	Raw

It is called RSdoc because is a combination of JAX-RS annotations [2] and Javadoc [22] (Illustraed in Figure 11).

Figure 11 RSdoc, JAX-RS and Javadoc



Swagger UI - Google Chrome

https://[SDN_CONTROLLER_ADDRESS]:8443/api

RSdoc — Health Monitor v1.0 — X-Auth-Token — Explore

/switches — Show/Hide | List Operations | Expand Operations | Raw

GET	/switches	List all Switch items
POST	/switches	Adds a Switch
GET	/switches/{id}	Get the specified Switch by its identifier
PUT	/switches/{id}	Updates a Switch

Implementation Notes

Normal Response Code(s): ok (200)

Error Response Codes: badRequest (400), unauthorized (401), forbidden (403), badMethod (405), serviceUnavailable (503)

Parameters

Parameter	Value	Description
id	(required)	the Switch identifier.
request	(required)	JSON representation of the Switch with updated data.

Try it out!

DELETE	/switches/{id}	Deletes a Switch

```
...
@Path("switches")
public class SwitchResource extends ControllerResource {

    ...

    /**
     * Updates a <type>Switch</type>.
     * <p>
     * Normal Response Code(s): ok (200)
     * <p>
     * Error Response Codes: badRequest (400), unauthorized (401), forbidden
     * (403), badMethod (405), serviceUnavailable (503)
     *
     * @param id the <type>Switch</type> identifier.
     * @param request JSON representation of the <type>Switch</type> with
     *        updated data.
     * @return a JSON encoding of the updated <type>Switch</type>.
     */
    @PUT
    @Path("{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response update(@PathParam("id") long id, String request) {
        ...
    }

    ...
}
```

JAX-RS annotations and Javadoc are already written when implementing RESTful Web Services, and they are re-used to 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 160 for an example.

Rsdoc Live Reference

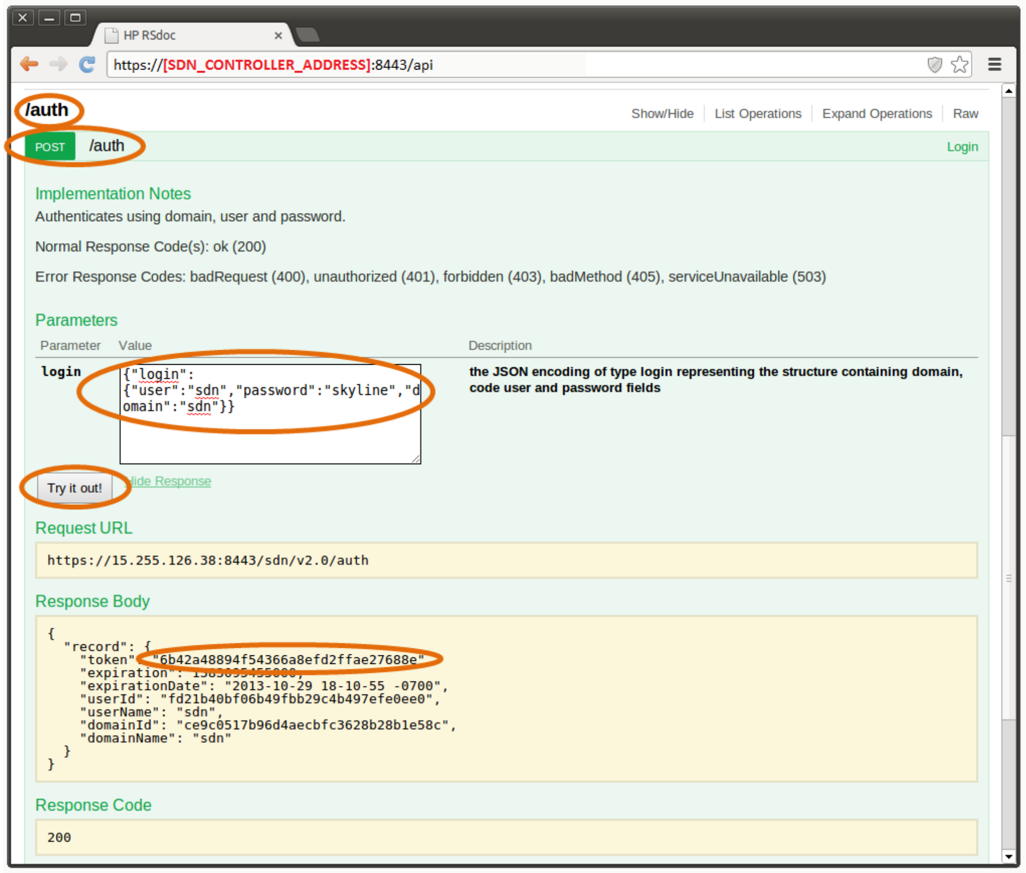
To access the HP VAN SDN Controller's Rsdoc (including extensions by applications):

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.

Figure 12 Authenticating via RSdoc Step 1



Implementation Notes

Authenticates using domain, user and password.

Normal Response Code(s): ok (200)

Error Response Codes: badRequest (400), unauthorized (401), forbidden (403), badMethod (405), serviceUnavailable (503)

Parameters

Parameter	Value	Description
login	`{"login":{"user":"sdn","password":"skyline","domain":"sdn"}}`	the JSON encoding of type login representing the structure containing domain, code user and password fields

Request URL

```
https://15.255.126.38:8443/sdn/v2.0/auth
```

Response Body

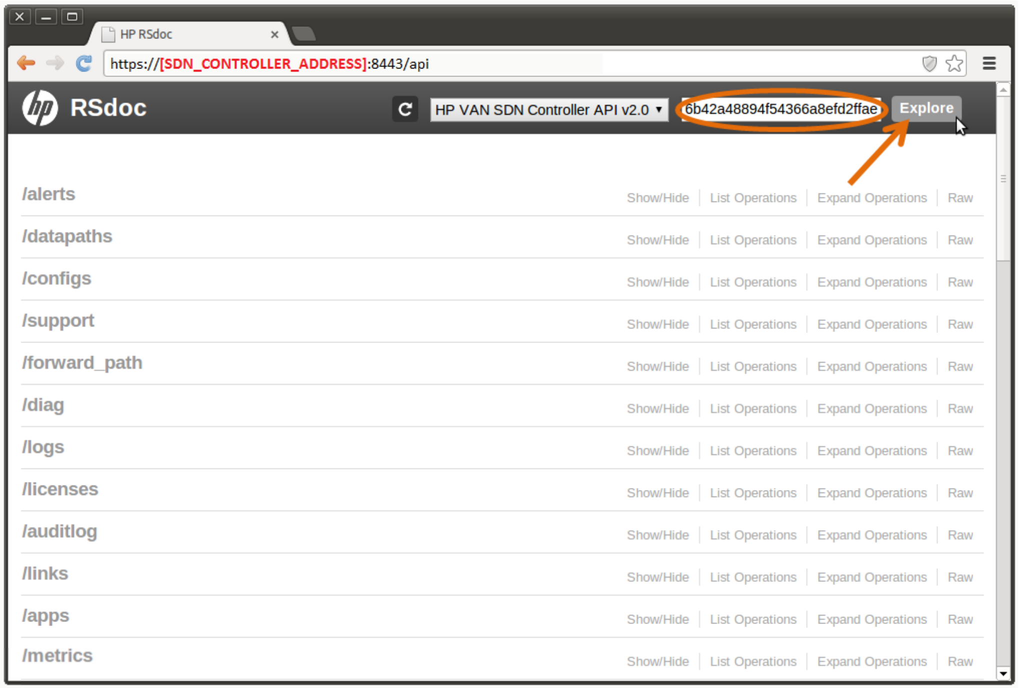
```
{
  "record": {
    "token": "6b42a48894f54366a8efd2ffae27688e"
    "expiration": 1383095455000,
    "expirationDate": "2013-10-29 18-10-55 -0700",
    "userId": "fd21b40bf06b49fbb29c4b497efe0ee0",
    "userName": "sdn",
    "domainId": "ce9c0517b96d4aecbfc3628b28b1e58c",
    "domainName": "sdn"
  }
}
```

Response Code

```
200
```

3. Set the authentication token as the *X-AUTH-TOKEN* in the RSdoc as illustrated in Figure 13. From this point all requests done via RSdoc will be authenticated as long as the token is valid.

Figure 13 Authenticating via RSdoc Step 2


HP RSdoc
https://[SDN_CONTROLLER_ADDRESS]:8443/api
RSdoc
HP VAN SDN Controller API v2.0
6b42a48894f54366a8efd2ffae
Explore

/alerts	Show/Hide	List Operations	Expand Operations	Raw
/datapaths	Show/Hide	List Operations	Expand Operations	Raw
/configs	Show/Hide	List Operations	Expand Operations	Raw
/support	Show/Hide	List Operations	Expand Operations	Raw
/forward_path	Show/Hide	List Operations	Expand Operations	Raw
/diag	Show/Hide	List Operations	Expand Operations	Raw
/logs	Show/Hide	List Operations	Expand Operations	Raw
/licenses	Show/Hide	List Operations	Expand Operations	Raw
/auditlog	Show/Hide	List Operations	Expand Operations	Raw
/links	Show/Hide	List Operations	Expand Operations	Raw
/apps	Show/Hide	List Operations	Expand Operations	Raw
/metrics	Show/Hide	List Operations	Expand Operations	Raw

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 changes system. The subsystem comprises of 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.

- 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 Audit Log Entry DTO 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 185.

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 ant 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 181.

Configuration

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 [23] [24] and MetaType [25] [26] 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 [27] and Maven SCR [28] plugin in a Maven *pom.xml* file for the application (See Root POM File on page 108). The SCR annotations must be included as a dependency, and the SCR plug-in is a build plugin.

Application pom.xml Example:

```
<?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>
    <build>
        <plugins>
            ...
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-scr-plugin</artifactId>
                <version>1.13.0</version>
                <executions>
                    <execution>
                        <id>generate-scr-srcdescriptor</id>
                        <goals>
                            <goal>scr</goal>
                        </goals>
                    </execution>
                </executions>
                <configuration>
                    <supportedProjectTypes>
                        <supportedProjectType>bundle</supportedProjectType>
                        <supportedProjectType>war</supportedProjectType>
                    </supportedProjectTypes>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

The component can then use Annotations to define the configuration properties as illustrated in the following listing.

Configurable Property Key Definition Example:

```
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:

```
...
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:

```
    ...
    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:

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

OpenFlow

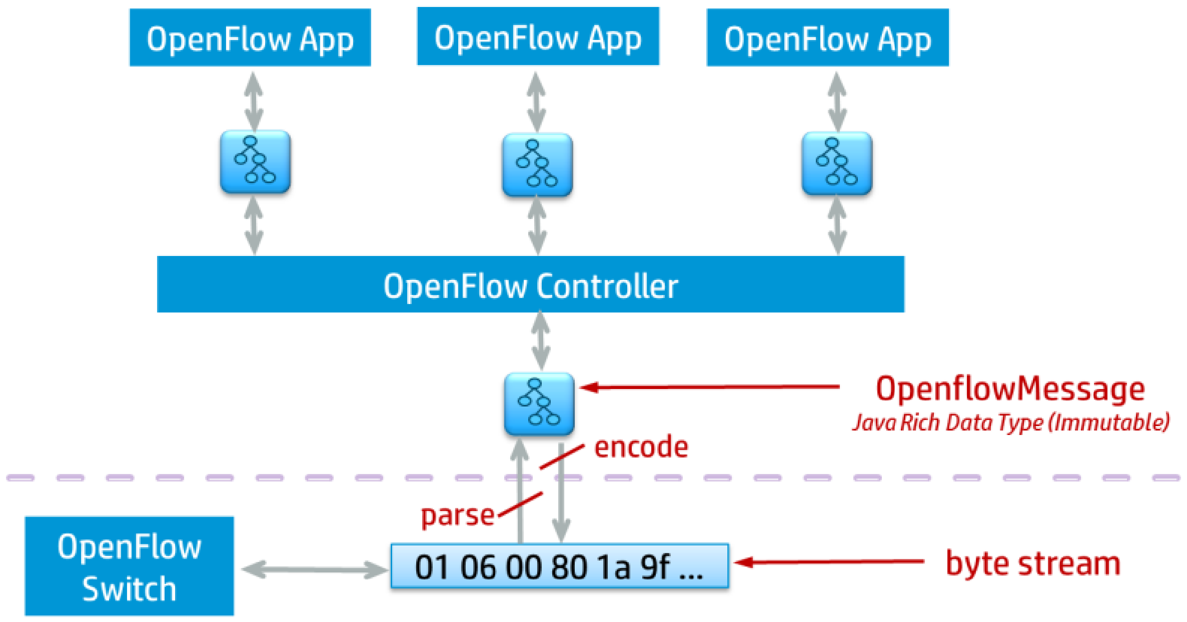
OpenFlow messages are sent and received between the controller and the switches (datapaths) it manages. These messages are byte streams, the structure of which is documented in the OpenFlow Protocol Specification documents published by the Open Networking Foundation (ONF) [29].

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 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.

This 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:

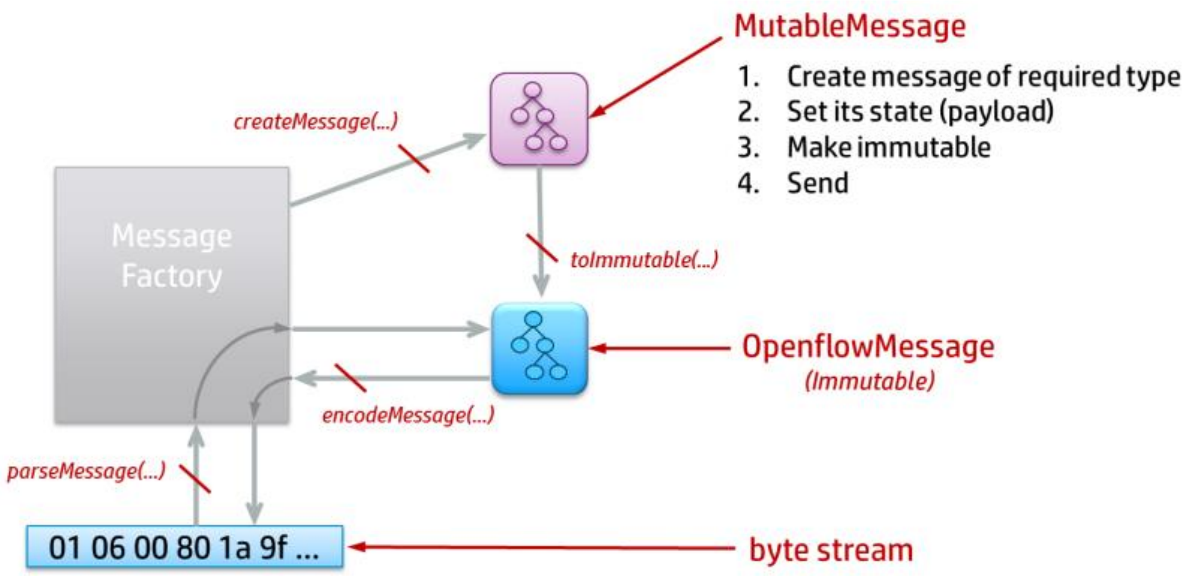
- To span all protocol versions
  - However, actively supporting just 1.0.0 and 1.3.1
- To be extensible
  - Easily accommodating future versions
- To provide an elegant, yet simple, API for dealing with OpenFlow messages
- To reduce the burden on application developers
  - Insulating developers from 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 will be fully creatable/encodable/decodable, making the library completely symmetrical in this respect.
  - The controller (or app) would never create 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 will facilitate the writing of automated tests to verify switch/controller interactions in a deterministic manner.
- Message instances, for the most part, will be immutable.
  - This means a single instance may 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, "Cacheable 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 will be defined with a constant for each bit.
  - A specific bitmask value will be represented by a Set of the appropriate enumeration constants.
  - For example: Set<PortConfig>
- A message instance will be mutable only while the message is under construction (for example, an application composing a FlowMod message). To be sent through the system is 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.

Figure 15 Message Factory Role



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:

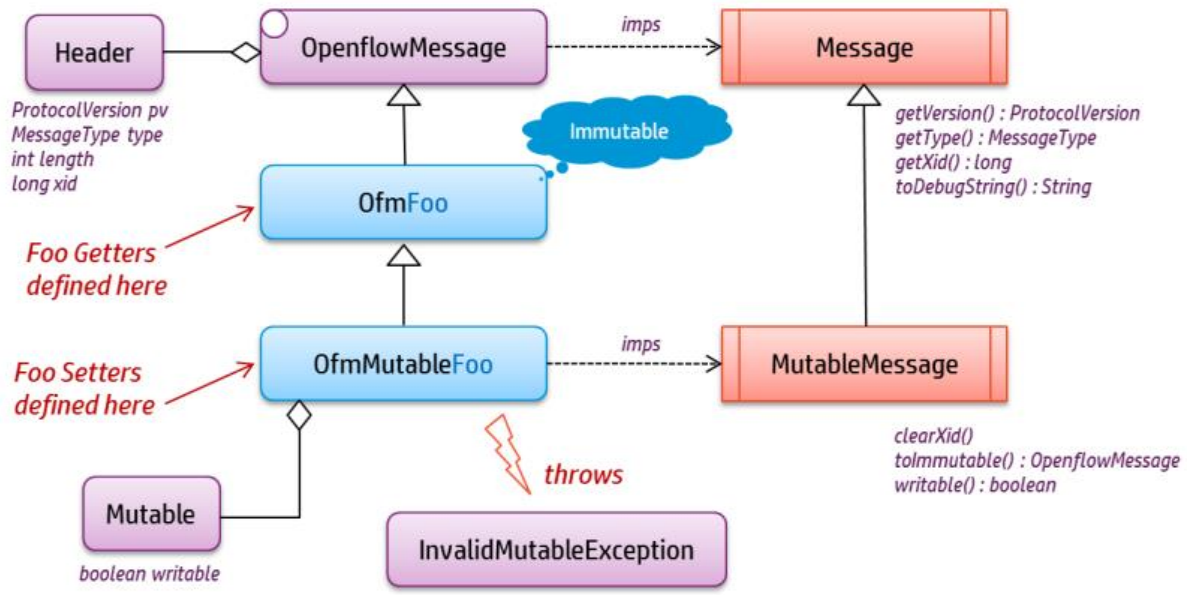
- The protocol version
- The message type
- The message length (in bytes)
- The transaction ID (XID)

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 cacheable 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:

Figure 16 OpenFlow Message Class Diagram



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 immutable 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.

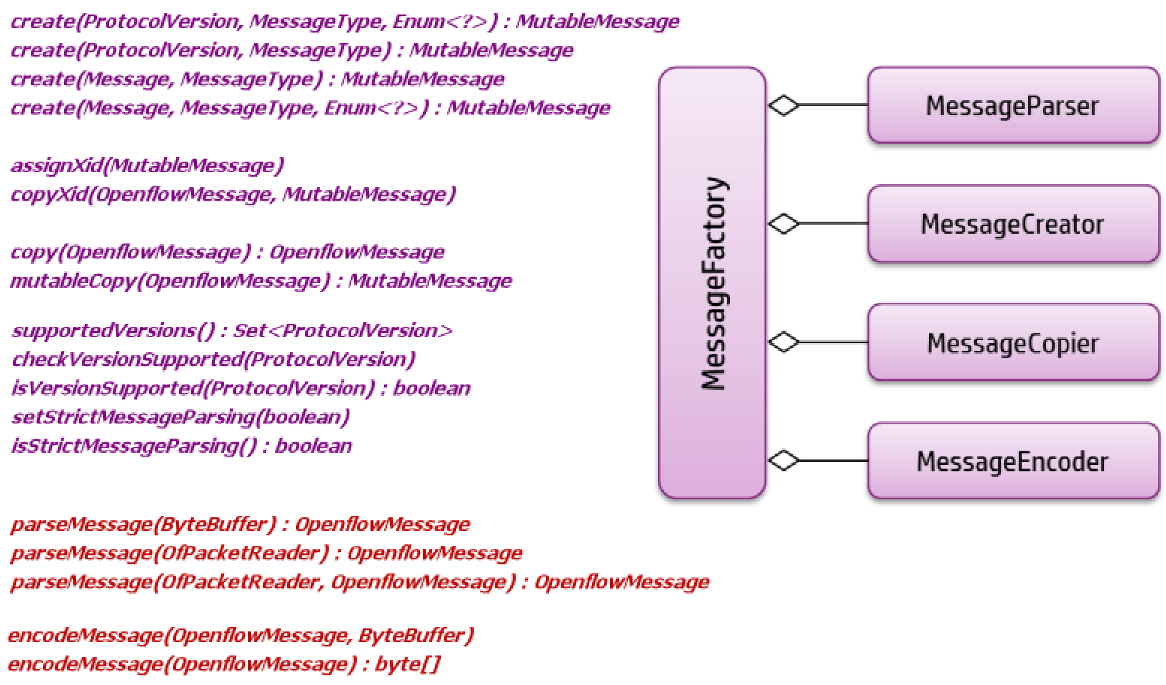
Note also that structures (e.g. a Match) have a very similar class relationship.

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*:

Figure 17 Message Factory Class Diagram


create(ProtocolVersion, MessageType, Enum<?>) : MutableMessage
create(ProtocolVersion, MessageType) : MutableMessage
create(Message, MessageType) : MutableMessage
create(Message, MessageType, Enum<?>) : MutableMessage
assignXid(MutableMessage)
copyXid(OpenflowMessage, MutableMessage)
copy(OpenflowMessage) : OpenflowMessage
mutableCopy(OpenflowMessage) : MutableMessage
supportedVersions() : Set<ProtocolVersion>
checkVersionSupported(ProtocolVersion)
isVersionSupported(ProtocolVersion) : boolean
setStrictMessageParsing(boolean)
isStrictMessageParsing() : boolean
parseMessage(ByteBuffer) : OpenflowMessage
parseMessage(OfPacketReader) : OpenflowMessage
parseMessage(OfPacketReader, OpenflowMessage) : OpenflowMessage
encodeMessage(OpenflowMessage, ByteBuffer)
encodeMessage(OpenflowMessage) : byte[]
MessageFactory
MessageParser
MessageCreator
MessageCopier
MessageEncoder

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:

Flowmod Message Example:

```
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 BufferId BUFFER_ID = BufferId.NO_BUFFER;
    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.

        fm.cookie(COOKIE).tableId(TABLE_ID).priority(FLOW_PRIORITY)
                .idleTimeout(FLOW_IDLE_TIMEOUT)
                .hardTimeout(FLOW_HARD_TIMEOUT)
                .bufferId(BUFFER_ID).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.FieldFactory.createBasicField;
        //   com.hp.of.lib.match.OxmBasicFieldType.*;

        MutableMatch mm = MatchFactory.createMatch(PV)
```

```
                .addField(createBasicField(PV, ETH_SRC, MAC, MAC_MASK))
                .addField(createBasicField(PV, ETH_TYPE, EthernetType.IPv4))
                .addField(createBasicField(PV, IP_PROTO, IpProtocol.TCP))
                .addField(createBasicField(PV, TCP_DST, SMTP_PORT));

        return (Match) mm.toImmutable();
    }

    private static final long INS_META_MASK = 0xffff0000;
    private static final long INS_META_DATA = 0x33ab0000;

    private List<Instruction> createInstructions() {
        // NOTE static imports of:
        //   com.hp.of.lib.instr.ActionFactory.createAction;
        //   com.hp.of.lib.instr.ActionFactory.createActionSetField;
        //   com.hp.of.lib.instr.InstructionFactory.createInstruction;
        //   com.hp.of.lib.instr.InstructionFactory.createMutableInstruction;

        List<Instruction> result = new ArrayList<Instruction>();

        result.add(createInstruction(PV, InstructionType.WRITE_METADATA,
                INS_META_DATA, INS_META_MASK));

        InstrMutableAction apply = createMutableInstruction(PV,
                InstructionType.APPLY_ACTIONS);
        apply.addAction(createAction(PV, ActionType.DEC_NW_TTL))
                .addAction(createActionSetField(PV, ETH_DST, MAC_DEST))
                .addAction(createActionSetField(PV, IPV4_DST, IP_DEST));
        result.add((Instruction) apply.toImmutable());

        return result;
    }
}
```

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 1.3.1 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:
    - Datapaths connecting or disconnecting
    - Messages received from datapaths
    - Packets received from datapaths (packet-in processing)
    - 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.
    - This will provide some level of protection for the controller and for the listeners, from an ill-performing listener implementation.
    - 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 queueing 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:
    - Core Controller—listens for connections from, and maintains state information about, OpenFlow switches (datapaths).
    - Packet Sequencer—listens for Packet-In messages, orchestrates the processing and subsequent transmission of Packet-Out replies.
    - 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`:

```
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

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

Datapath Information Example:

```
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 listen for *ECHO_REPLY* messages (presumably we have some other code that is sending *ECHO_REQUEST* messages), and *PORT_STATUS* messages.

ECHO_REPLY and PORT_SATUS Example:

```
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;
            }
        }
    }

    private void handleEchoReply(OfmEchoReply msg, DataPathId dpid) {
        ...
    }

    private void handlePortStatus(OfmPortStatus msg, DataPathId dpid) {
        ...
    }
}
```

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:

```
    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().

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:

```
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);
        long now = System.currentTimeMillis();
        long then = retrieveTimestamp(future.reply());
        long duration = now - then;
        log.info"ECHO Latency to {} is {} ms", dpid, duration);
    } catch (Exception e) {
        log.warn(E_ECHO_FAILED, e.toString());
    }
}
```

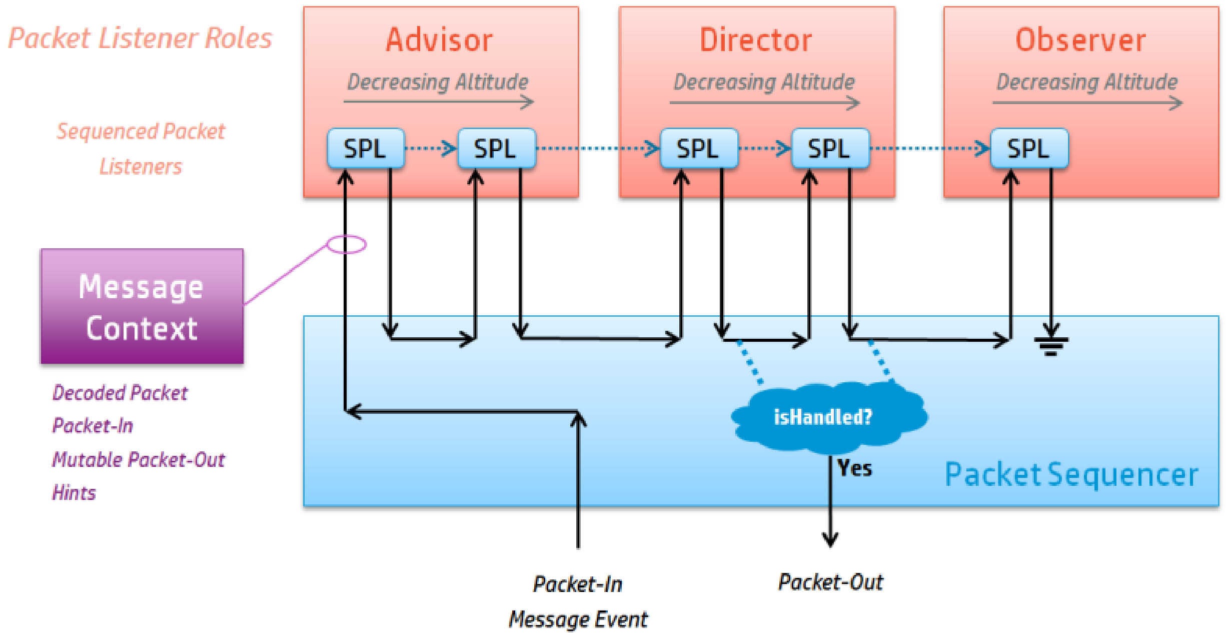
```
private OpenflowMessage createEchoRequest(byte[] timestamp) {
    OfmMutableEchoRequest echo = (OfmMutableEchoRequest)
            MessageFactory.create(PV, MessageType.ECHO_REQUEST);
    echo.data(timestamp);
    return echo.toImmutable();
}

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:

Figure 18 Packet-In Processing



The Roles proide 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:
    - The return value from an ADVISOR's *event()* callback is ignored (but should be *false* to be polite).
    - The return value from a DIRECTOR's *event()* callback should be *true* if it is determined that the packet has been "handled" (i.e. the PACKET_OUT message is ready to send); *false* otherwise.
    - The return value from DIRECTORs and OBSERVERs downstream of the DIRECTOR that "handled" the packet is ignored. (OBSERVERs should return *false* to be polite).
- Once a DIRECTOR returns *true* 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:

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

private final MyObserver myObserver = new MyObserver();

private void register() {
    cs.addPacketListener(myObserver, PacketListenerRole.OBSERVER,
            OBS_ALTITUDE, OBS_INTEREST);
}


private static class MyObserver implements SequencedPacketListener {
    @Override
    public boolean event(MessageContext context) {
        Dns dns = context.decodedPacket().get(ProtocolId.DNS);
```

```
            reportOnDnsPacket(dns, context.srcEvent().dpid());
            return false;
        }

        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.
            // ...
        }

        @Override
        public void errorEvent(ErrorEvent event) {
            // Never gets called for Observers
        }
    }
```

- 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.
- *decodedPacket()* – returns the network packet model (immutable) of the decoded packet data.
- *getProtocols()* – returns an array 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 the PACKET_OUT message.
- *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.

Metrics Framework

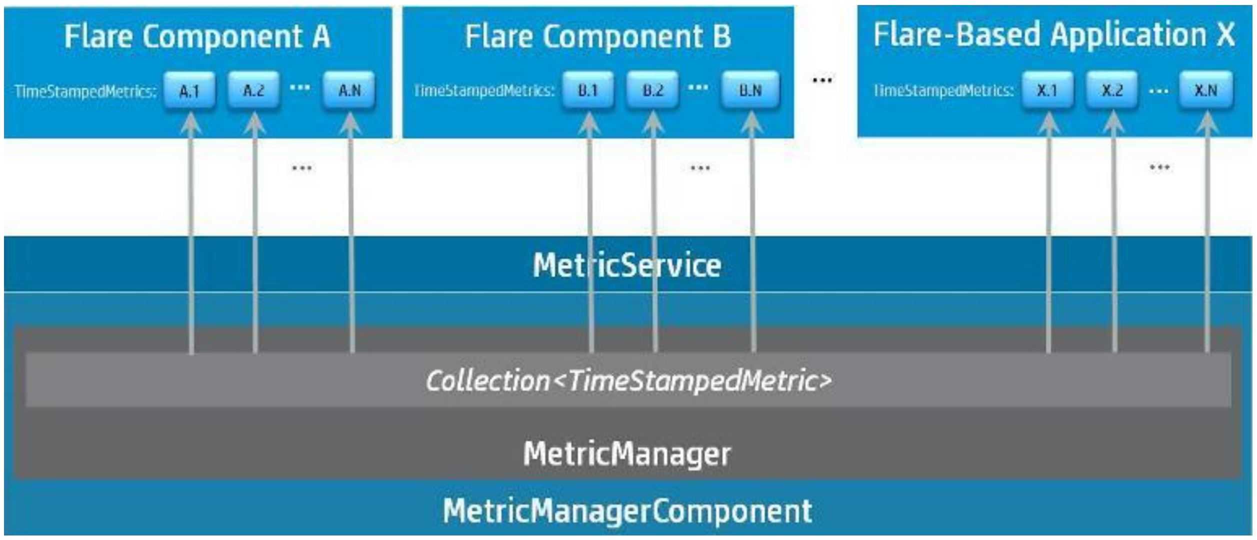
The fundamental objectives to be addressed by the metering framework are as follows.

- Support components that are part of the HP VAN SDN Controller Framework and applications that aren't.
- Make metrics simple to use.
- Support the creation and updating of metrics within the controller and from outside, to accommodate apps those 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.
- Be robust.
    - Survive a scenario in which the controller bounces.
    - Survive a scenario in which the metering framework bounces but the controller doesn't.
- 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


Flare Component A
TimeStampedMetrics: A.1 A.2 ... A.N
Flare Component B
TimeStampedMetrics: B.1 B.2 ... B.N
...
Flare-Based Application X
TimeStampedMetrics: X.1 X.2 ... X.N
MetricService
Collection<TimeStampedMetric>
MetricManager
MetricManagerComponent

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 *TimeStampedMetrics* In 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 *TimeStampedMetrics*.

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 *TimeStampedMetrics* over time may read via 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 quantile 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. TheMetricDescriptor 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 mostTimeStampedMetrics 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 1 Metric Descriptor Sybtype

TimeStampedMetric **Subtype**	**Corresponding** ***MetricDescriptor*** **Subtype**	**Required** ***MetricDescriptorBuilder*** **Subtype**
TimeStampedCounter	*CounterDescriptor*	*CounterDescriptorBuilder*
TimeStampedGauge	*GaugeDescriptor*	*GaugeDescriptorBuilder*
TimeStampedHistogram	*HistogramDescriptor*	*HistogramDescriptorBuilder*
TimeStampedMeter	*MeterDescriptor*	*MeterDescriptorBuilder*
TimeStampedRatioGauge	*RatioGaugeDescriptor*	*RatioGaugeDescriptorBuilder*
TimeStampedRollingCounter	*RollingCounterDescriptor*	*RollingCounterDescriptorBuilder*

TimeStampedTimer	*TimerDescriptor*	*TimerDescriptorBuilder*

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 theMetricDescriptorBuilder 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 oneTimeStampedMetric type, TimeStampedRollingCounter, offers the option to specify an extra parameter above and beyond the properties conveyed by theMetricDescriptor object.

MetricService:

```
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 aTimeStampedMetric 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 aTimeStampedMetric, 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 theTimeStampedMetric 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 theirTimeStampedMetrics 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 forTimeStampedMetrics whose values are persisted. But for application components outside the controller that created and are updating TimeStampedMetricsthere may be no indication that the controller has bounced - or gone down and stayed down - until the next time they try to update TimeStampedMetricvalues.

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 usingTimeStampedMetrics 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 TimeStampedMetricsthere 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 TimeStampedMetricthat 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:

```
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 span 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 theTimeStampedMetric 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 bring 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, twelves "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.

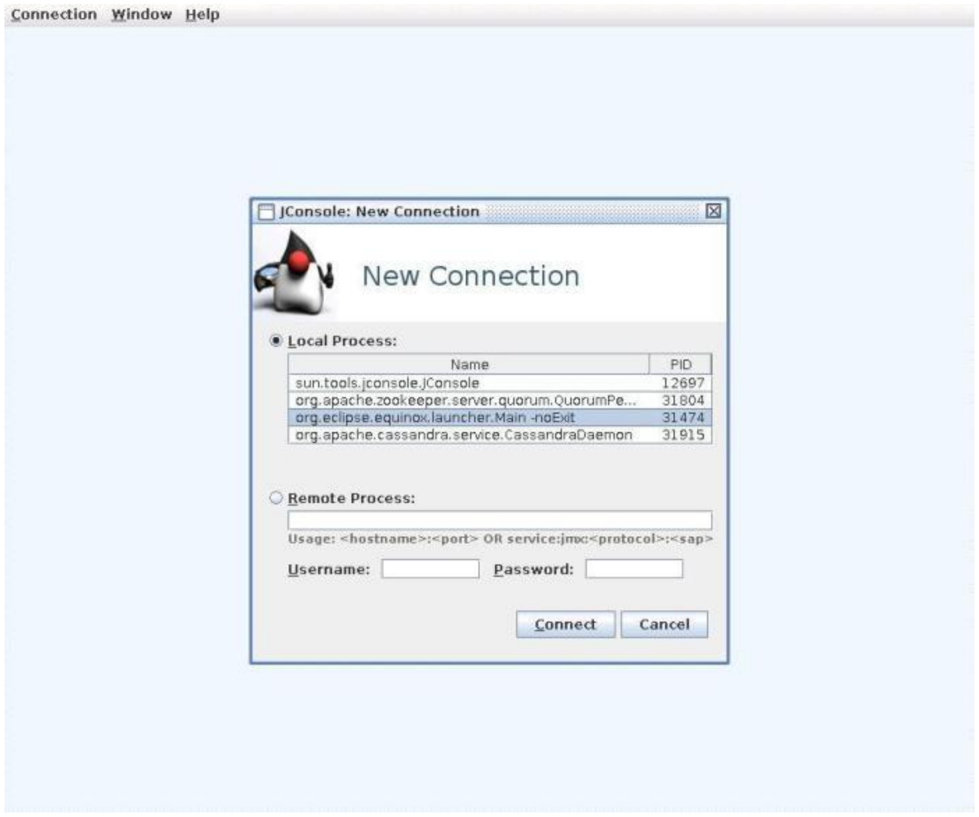
JMX Clients

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 eachTimeStampedMetric. 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 theTimeStampedMetric 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

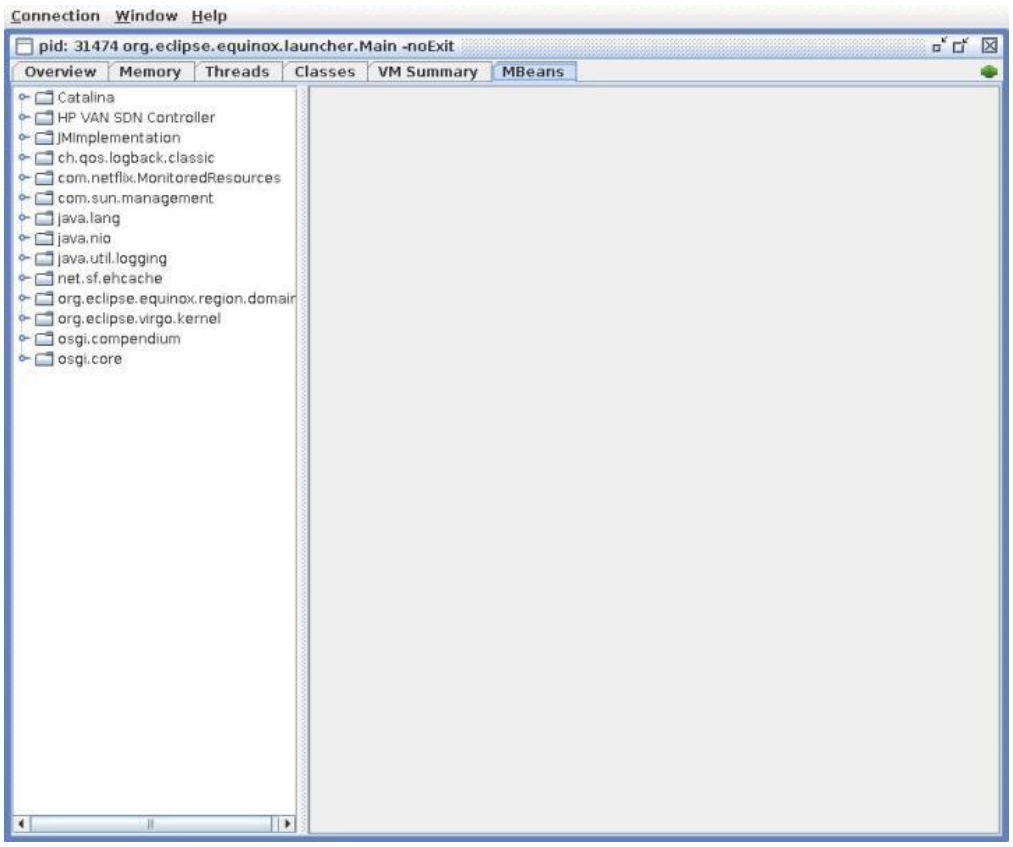


Name	PID
sun.tools.jconsole.JConsole	12697
org.apache.zookeeper.server.quorum.QuorumPe...	31804
org.eclipse.equinox.launcher.Main -noExit	31474
org.apache.cassandra.service.CassandraDaemon	31915

Usage: <hostname>:<port> OR service:jmx:<protocol>:<sap>

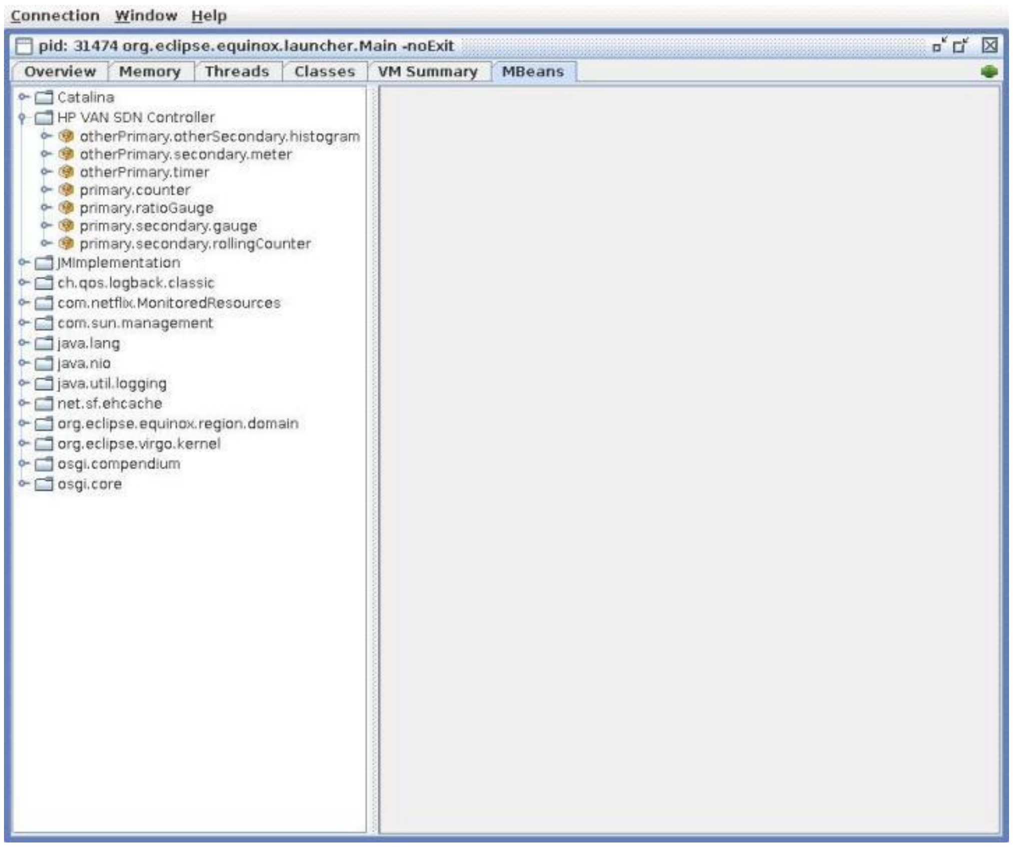
Choose a local connection to the JMX server instance that looks like the one highlighted in the preceding screenshot and click the Connect button. Upon successfully connecting to that JMX server instance, one should see a screen that looks something like Figure 21.

Figure 21 JConsole



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).

Figure 22 JConsole – HP VAN SDN Controller Metrics



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.

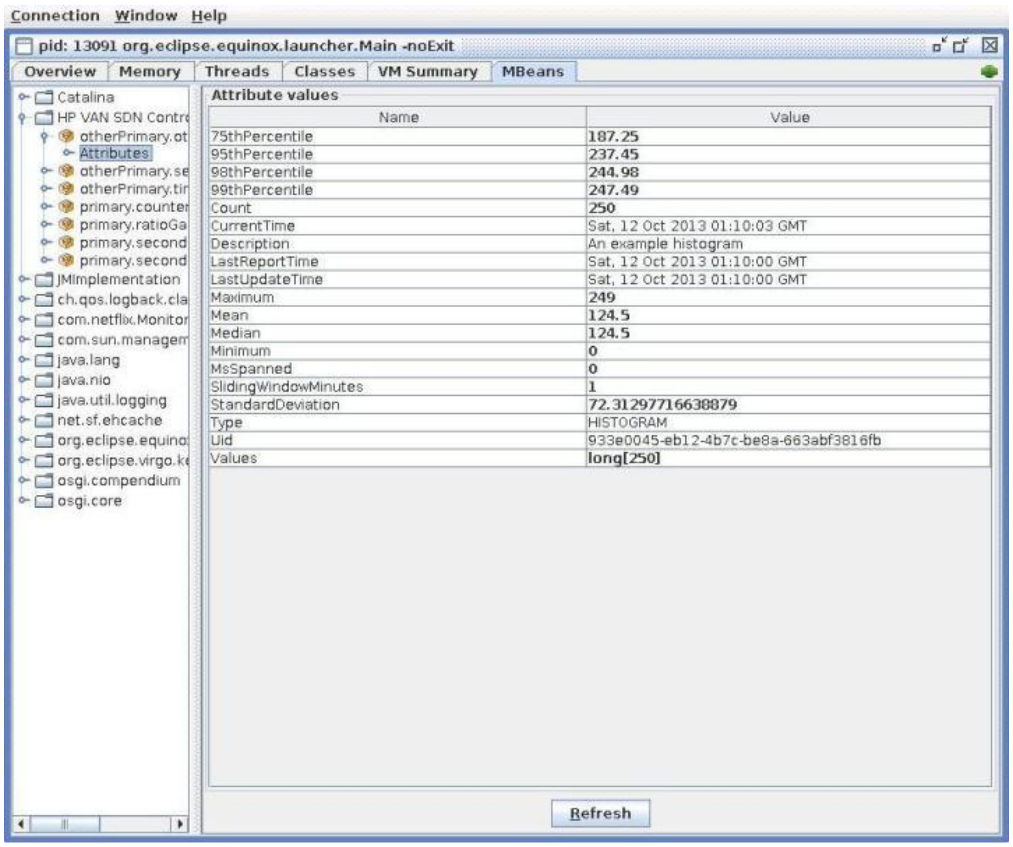


Figure 23 JConsole – Metric Example

Name	Value
75thPercentile	187.25
95thPercentile	237.45
98thPercentile	244.98
99thPercentile	247.49
Count	250
CurrentTime	Sat, 12 Oct 2013 01:10:03 GMT
Description	An example histogram
LastReportTime	Sat, 12 Oct 2013 01:10:00 GMT
LastUpdateTime	Sat, 12 Oct 2013 01:10:00 GMT
Maximum	249
Mean	124.5
Median	124.5
Minimum	0
MsSpanned	0
SlidingWindowMinutes	1
StandardDeviation	72.31297716638879
Type	HISTOGRAM
Uid	933e0045-eb12-4b7c-be8a-663abf3816fb
Values	long[250]

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

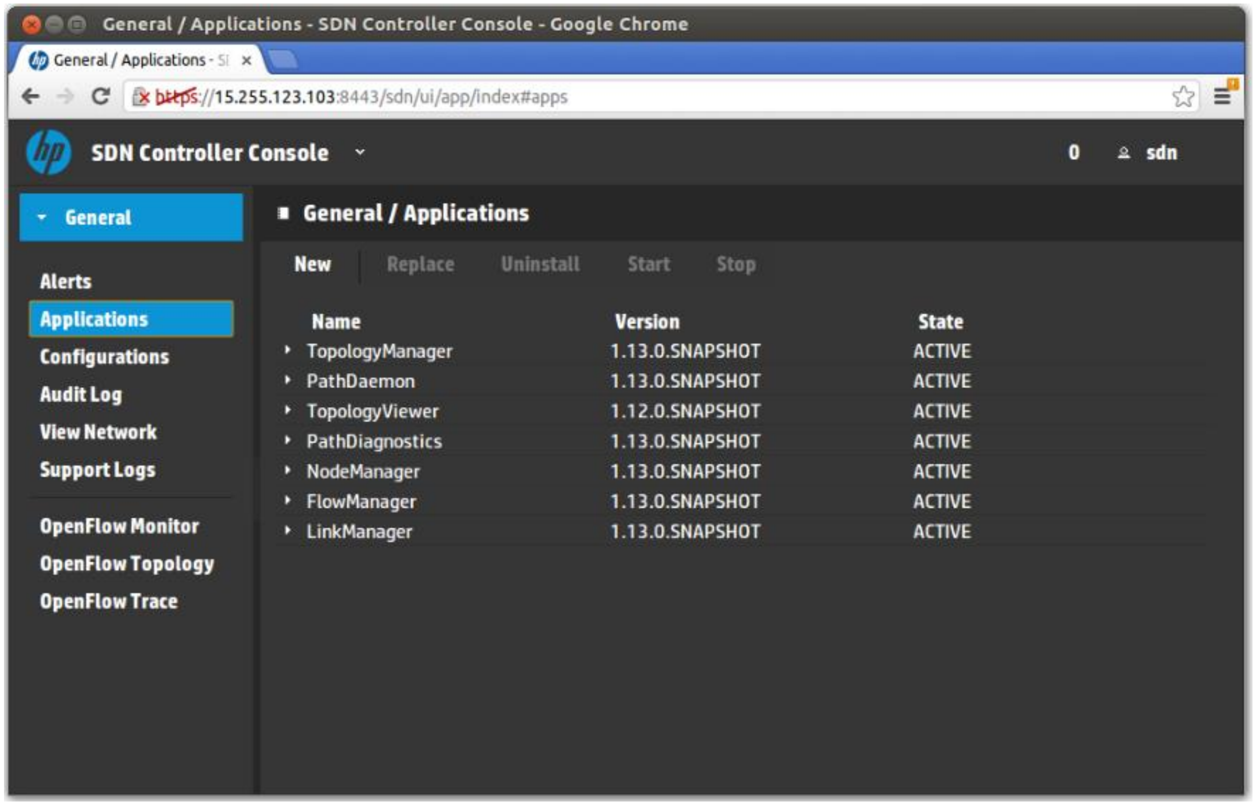
SKI Framework - Overview

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):
  - HTML Templates—providing alternate layouts for the UI
  - Core SKI Framework—providing navigation, search, and basic view functionality
  - Reference Documentation—documenting the core framework and library APIs
  - Reference Implementation—providing an example of how application code might be written
- SKI Assets (Server Side):
  - Java Classes—providing assistance in formulating RESTful Responses

Figure 24 SDN Controller main UI

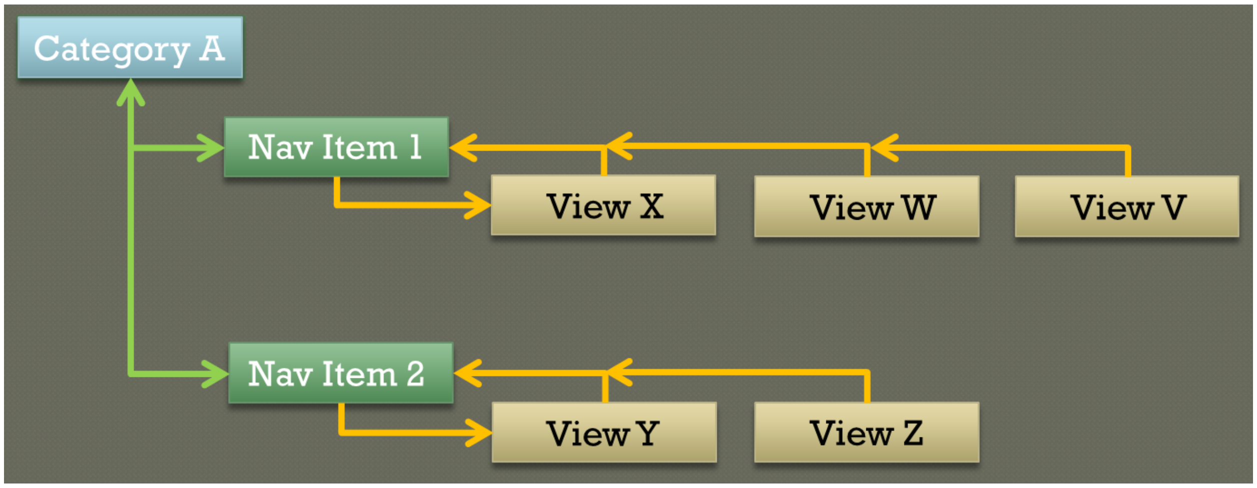


Name	Version	State
TopologyManager	1.13.0.SNAPSHOT	ACTIVE
PathDaemon	1.13.0.SNAPSHOT	ACTIVE
TopologyViewer	1.12.0.SNAPSHOT	ACTIVE
PathDiagnostics	1.13.0.SNAPSHOT	ACTIVE
NodeManager	1.13.0.SNAPSHOT	ACTIVE
FlowManager	1.13.0.SNAPSHOT	ACTIVE
LinkManager	1.13.0.SNAPSHOT	ACTIVE

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.

Figure 25 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:

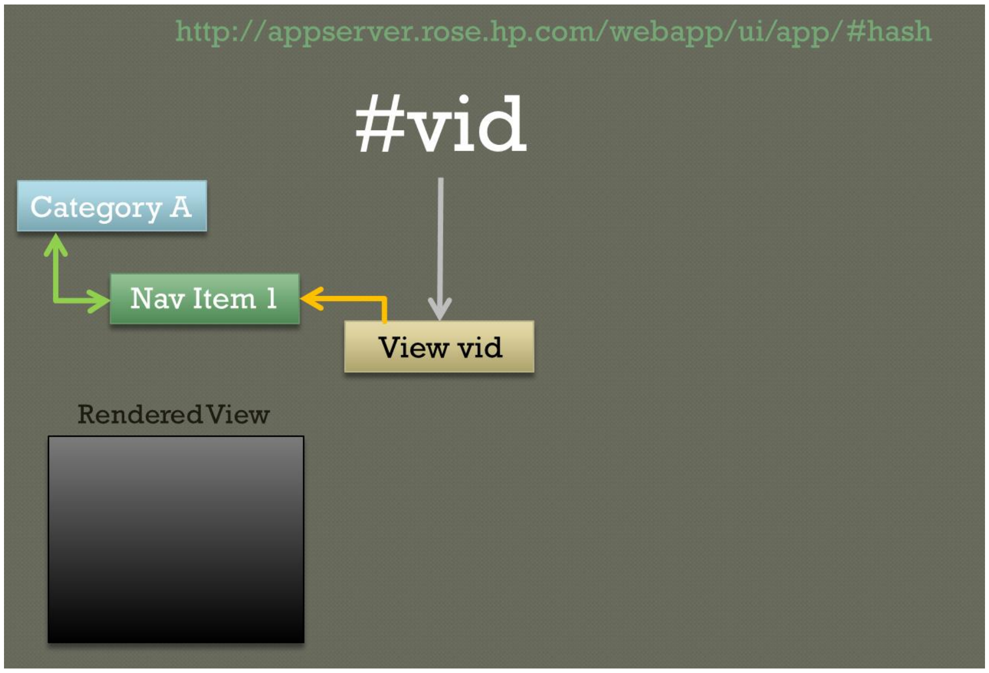
http://appserver.rose.hp.com/webapp/ui/app/#hash

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 with respect to the view (that is, select a specific row in a table)

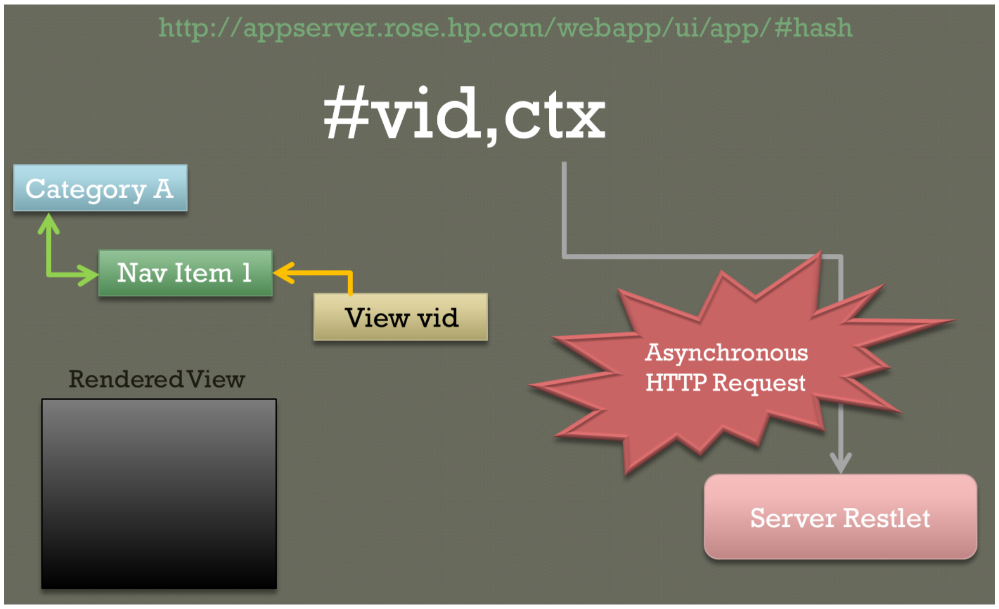
The following diagrams show the sequence of events on how SKI selects a view and loads the data if a URL is pasted into the browser. The *#hash* is decoded into *#vid,ctx,sub*, as shown in Figure 26. The vid (view ID) is used to determine the view, navigation item and category to be selected.

Figure 26 SKI UI view hash diagram



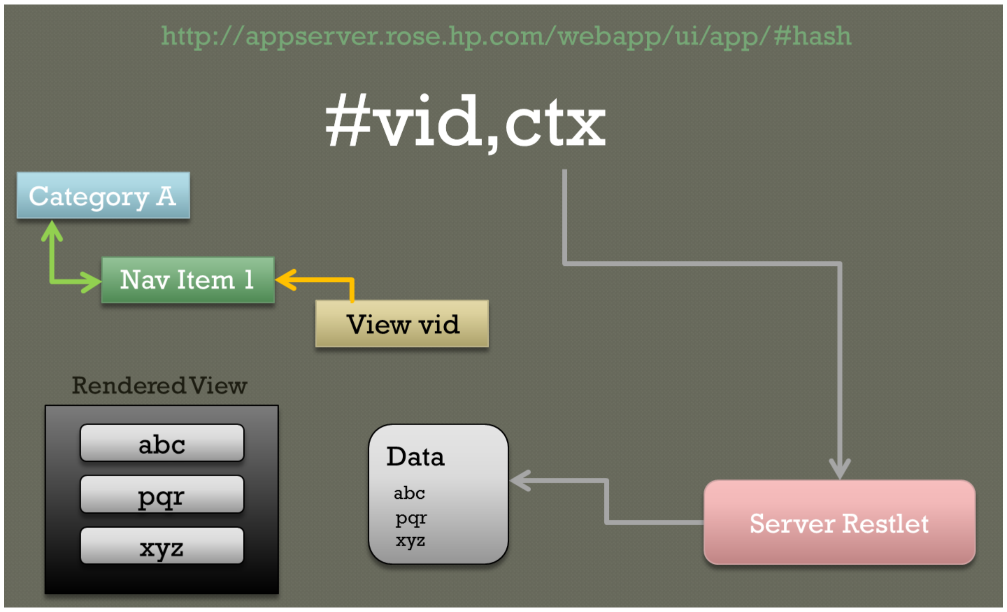
Next, the *ctx* (context), shown in Figure 27, can be used to help determine what data to retrieve from the Server RESTlet.

Figure 27 SKI UI view and context hash diagram


http://appserver.rose.hp.com/webapp/ui/app/#hash
#vid,ctx
Category A
Nav Item 1
View vid
Rendered View
Asynchronous HTTP Request
Server Restlet

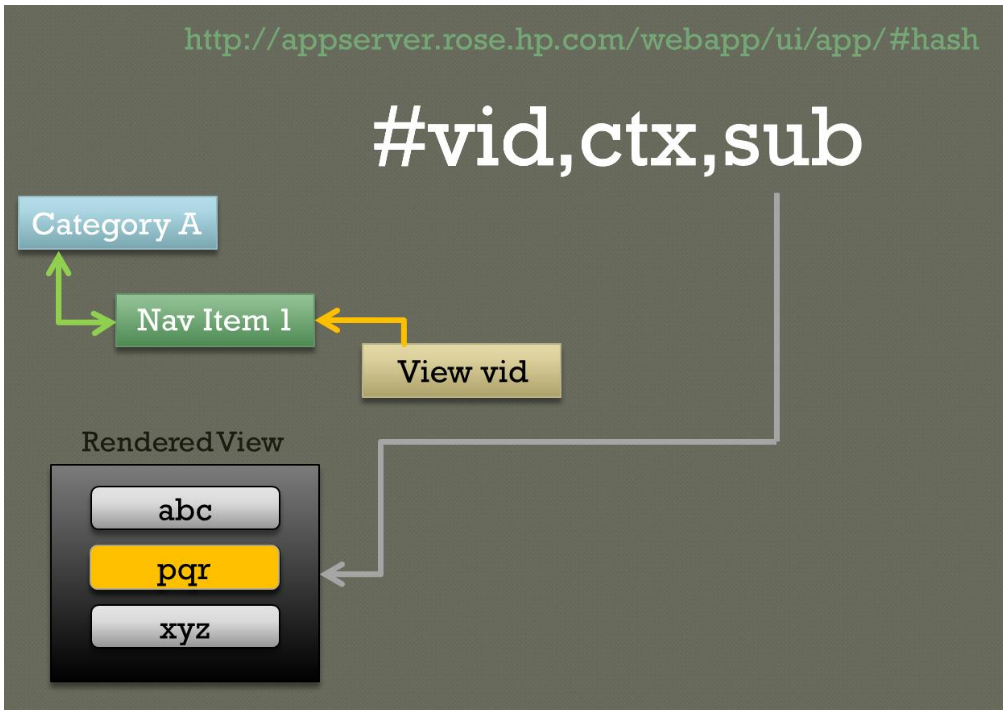
When the Asynchronous HTTP request returns, the data (likely in JSON form), as shown in Figure 28, can be used to populate the view's DOM (grids, widgets, etc.).

Figure 28 SKI UI view data retrieval diagram



Finally, the *sub* (**sub-context**) can be used to specify addition 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:

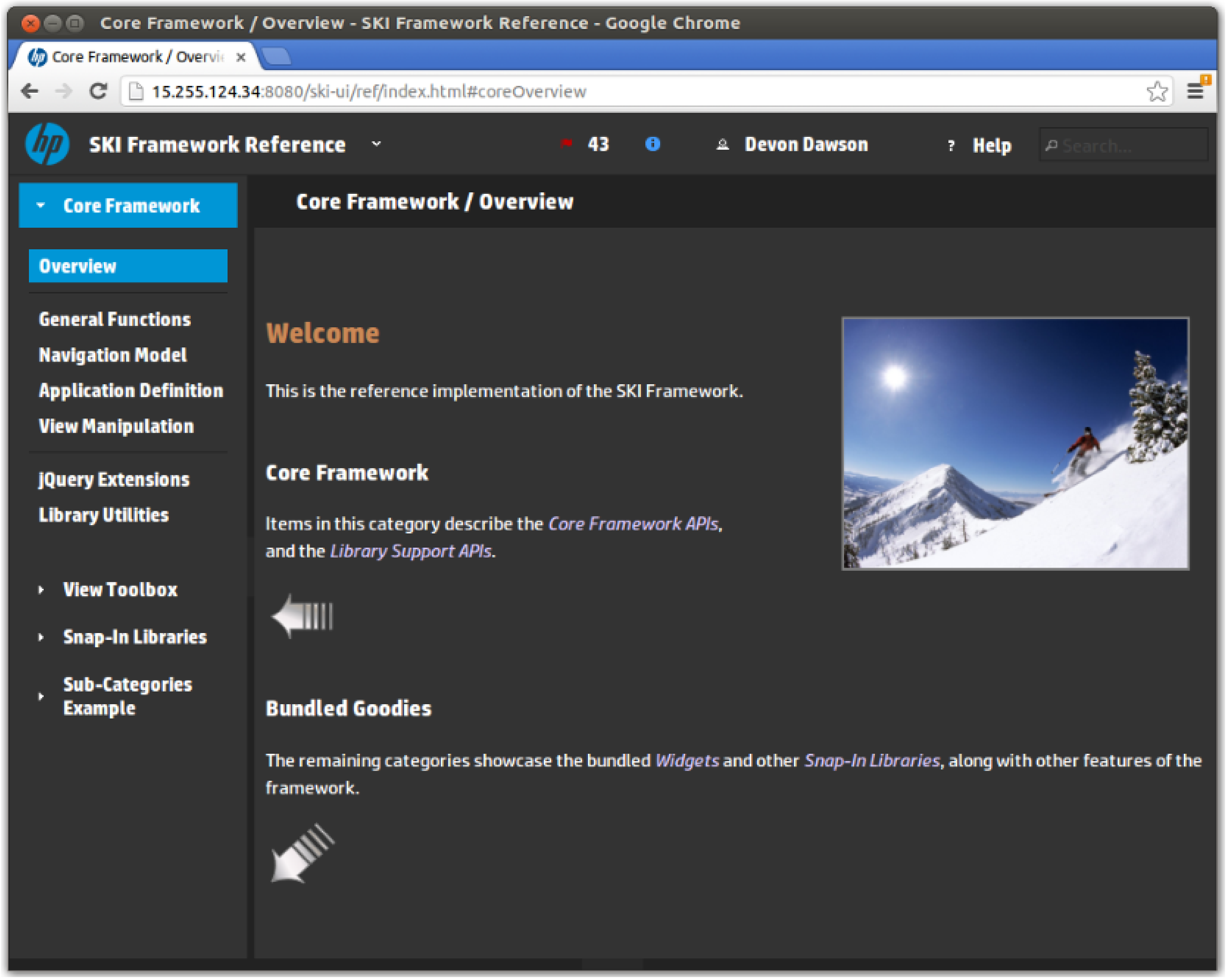
- **Create**—called a single time when the view needs to be created (that is, navigation item is clicked for the first time). At this time, a view will return its created DOM structure (that is, an empty table).
- **Preload**—called only once, after the view is in the DOM. At this time, a view can perform 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.

Figure 30 SKI UI reference application



Welcome

This is the reference implementation of the SKI Framework.

Core Framework

Items in this category describe the *Core Framework APIs*, and the *Library Support APIs*.

Bundled Goodies

The remaining categories showcase the bundled *Widgets* and other *Snap-In Libraries*, along with other features of the framework.

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 4 Sample Application on page 106.

Project: myapp
Directory: /myapp

File: pom.xml
Purpose: specifies the *Jersey* REST URL prefix

UI Extension webapp context path:

```
<properties>
  <jersey.version>1.8</jersey.version>
  <webapp.context>acme/ui/myapp</webapp.context>
```

Directory:**/myapp/src/main/java**
Javapackage:**com.acme.myapp.ui**
File:**MyAppUIExtension.java**
Purpose: creates the UI Extension registration class, providing SDN with the paths to find the *js.html* and *css.html* files.

When a UI project implements *ControllerUIExtension* (extending *SelfRegisteringUIExtension*), SDN will register the Extension to the SDN UI Extension Registry. The Registry keeps track of all available UI extensions. During runtime, when the main page (index) is loaded, the HTTP request for the index page is actually serviced by the *AppIndexResource*. This resource dynamically builds up the main HTTP page using a static template file (com.hp.sdn.ui.index.html) and injects all content from all of the registered UI Extensions by replacing embedded tags (CSS-INCLUDES, JAVASCRIPT-INCLUDES, and so on).

UI Extension Java implementation:

```
public class MyAppUIExtension extends SelfRegisteringUIExtension {
    /** Create the controller UI elements contributor. */
    public MyAppUIExtension() {
        super("myapp", "com/acme/myapp/ui", MyAppUIExtension.class);
    }
}
```

Directory: **/myapp/src/main/java**
Java package: **com.acme.myapp.ui.rs**
File: **MyAppResource.java**
Purpose: creates the server-side resource that returns JSON string data via the HTTP GET at URL *acme/ui/myapp/app/rs/mydata* (*app/rs* comes from the *web.xml*).

UI Extension Java resource implementation:

```
@Path("mydata")
public class MyAppResource extends ControllerResource {
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response data() {
        // code to create my JSON data
        return ok(myJsonData).build();
    }
}
```

Directory: **/myapp/src/main/resources**
Java package: **com.acme.myapp.ui**
File: **js.html**
Purpose: Javascript content to inject into the *index* HTML page generated by SDN (at the JAVASCRIPT-INCLUDES marker).

UI Extension Javascript code injection:

```
<script src="/acme/ui/myapp/app/my-nav.js"></script>
<script src="/acme/ui/myapp/app/my-view.js"></script>
```

Directory: **/myapp/src/main/resources**
Java package: **com.acme.myapp.ui**
File: **css.html**
Purpose: CSS content to inject into the *index* HTML page generated by SDN (at the CSS-INCLUDES marker).

UI Extension Javascript CSS injection:

```
<link href="/acmesdn/ui/ctl/app/ctl-style.css" rel="stylesheet">
```

Directory: **/myapp/src/main/resources**
Java package: **com.acme.myapp.ui.lion**
File: **nav-lion.properties**
Purpose: Java properties file containing the navigation item localization strings:

UI Extension navigation localization:

```
n-my-view = My Application
```

Directory: **/myapp/src/main/resources**
Java package: **com.acme.myapp.ui.lion**
File: **my-view.properties**
Purpose: Java properties file containing the view's localization strings.

UI Extension view localization:

```
Title = My Viewer
icon = grid
someText = This is some localized text.
```

Directory: **/myapp/src/main/webapp**
File: **my-nav.js**
Purpose: specifies where the view should be located in the SDN navigation model, the following is specifying that the navigation item *n-my-view* should be added after the *n-audits* navigation item in the SDN navigation tree, the navigation item selects view *my-view*.

UI Extension JavaScript navigation implementation:

```
(function (api) {
    'use strict';
```

```
        var f = api.fn,          // general functions API
            nav = api.nav;       // navigation model API

        // Add a new item to an existing category
        nav.insertItemsAfter('n-audits', [
            nav.item('n-my-view', 'my-view')
        ]);
    }(SKI));
```

Directory: **/myapp/src/main/webapp**
File: **my-app.js**
Purpose: defines the view content, handling the SKI framework life-cycle events.

UI Extension JavaScript View implementation:

```
    (function (api) {
        'use strict';
        //framework APIs
        var f = api.fn,     //general API
            def = api.def,  //application definition API
            t = api.lib.htmlTags,
            wgt = api.lib.widgetFactory,
            v = api.view;   //view API
        // create my content
        function myAppCreate(view) {
            // return my DOM content structure
        }
        // load my content
        function myAppLoad(view) {
            // update my DOM content from my ajax success call
        }
        def.addView('my-view', {
            create: myAppCreate,
            load: myAppLoad,
        });
    }(SKI));
    (function (api) {
        'use strict';
        //framework APIs
        var f = api.fn,     //general API
            def = api.def,  //application definition API
            t = api.lib.htmlTags,
            wgt = api.lib.widgetFactory,
            v = api.view;   //view API
        // create my content
        function myAppCreate(view) {
            // return my DOM content structure
        }
        // load my content
        function myAppLoad(view) {
            // update my DOM content from my ajax success call
        }
        def.addView('my-view', {
            create: myAppCreate,
            load: myAppLoad,
        });
    }(SKI));
```

Directory: **/myapp/src/main/webapp/WEB-INF**
File: **web.xml**
Purpose: defines the Java server-side resources for your application, note that the full URL path to the resource is *acme/ui/myapp/app/rs/*.

UI Extension web.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app        version="2.5"        xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <display-name>MyApp 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>
```

```
                    com.acme.myapp.ui.rs.MyAppResource
                    com.hp.sdn.rs.misc.AuthenticationHandler
                    com.hp.sdn.rs.misc.NotFoundErrorHandler
                </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>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>
```

Distributed HA Primitives

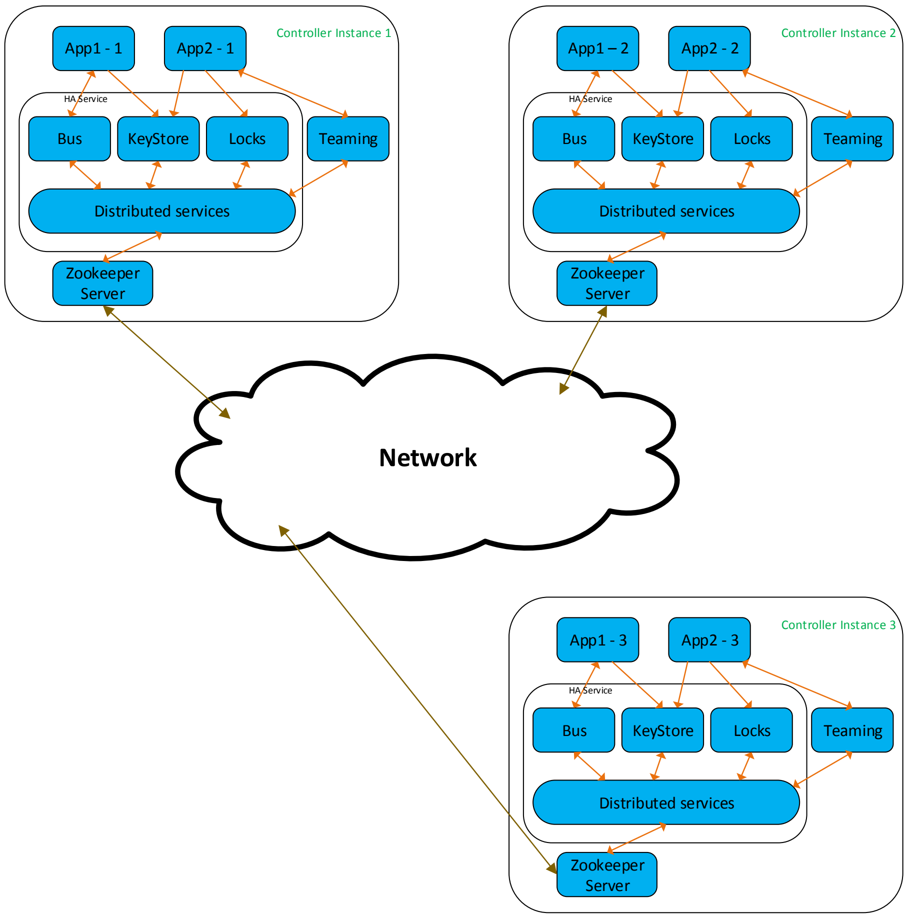
Introduction

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

- Controller Teaming
- HA service

Following figure describes the communication between the controller applications and the HP VAN SDN Controller HA sub-systems. "App1 – 1" indicates instance of application 1 on controller instance 1. Distributed services along with zookeeper, ensures the data synchronization across the controller cluster nodes.

Figure 31 Application view of HA Services



Controller Teaming

At broader level controller teaming provides 2 functionalities to the administrator and the controller applications.

- Teaming Configuration Service
- Teaming Service

Teaming Configuration Service

Teaming configuration service provides REST interfaces that can be used to setup controller node teaming. Without team configuration, controller nodes will bootstrap in standalone mode. As the teaming is configured, identified nodes forms a cluster and the controller Applications can communicate across the cluster using HA service interfaces.

For team creation help and other configuration commands please refer to HP VAN SDN Controller Administrator Guide [30].

Following curl command is 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/status?forwardRequest=true  -ksS
```

Teaming Service

Teaming helps in grouping a set of mandatory applications and control the role of the controller node in the cluster based on the mandatory application availability status. Based on the teaming configuration a controller node joins the team when all the mandatory applications joins the team.

Following are the different types of teaming events that joined application might receive:

BECOME_SUSPENDED—Controller node is not part of the team yet. Stay in suspended mode

BECOME_LEADER—Controller node elected as a leader of the team

BECOME_MEMBER—Controller node became a team member

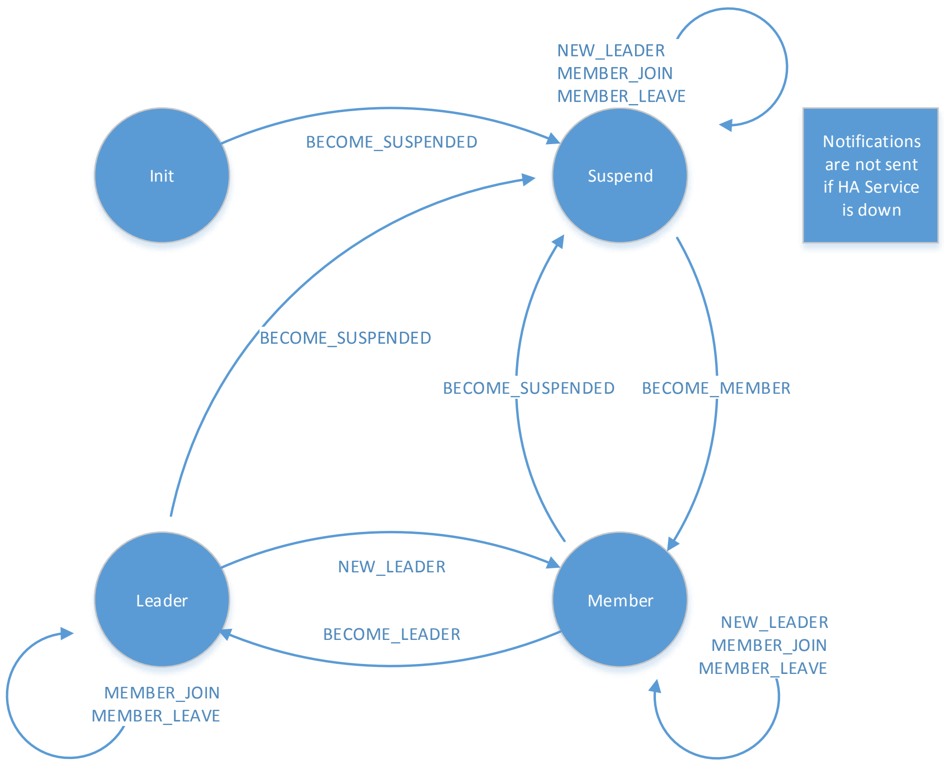
NEW_LEADER—Recent election resulted in a new team leader. Leader ID is notified through the event

MEMBER_JOIN—A new member node joined the team. Joined node ID is notified through the event

MEMBER_LEAVE—A member node left the team. Left node ID is notified through the event

Following diagram explains the state diagram of teaming service.

Figure 32 Teaming Service State Diagram



State Transitions

INIT to BECOME_SUSPENDED–On team Join, application receives SUSPEND notification

BECOME_SUSPEND to BECOME_MEMBER–Node is now part of the team and is a member entity

BECOME_MEMBER to BECOME_SUSPEND–Node is not part of the team now

BECOME_MEMBER to BECOME_LEADER–Re-election occurred within the team and the local node is elected the leader

BECOME_LEADER to BECOME_MEMBER–Re-election occurred within the team and the local node became a member. New leader detail are notified through NEW_LEADER event

BECOME_LEADER to BECOME_SUSPEND–Node is not part of the team now

Init sequence

First event application receives on controller team join is BECOME_SUSPEND. If the local node is not part of the team yet, until status change no nofifications are received by the application. Otherwise one of the following events occur

If the local node is the leader in the controller team application receives BECOME_MEMBER followed by BECOME_LEADER.

If the local node is the leader, application receives NEW_LEADER notification followed by BECOME_MEMBER.

Controller applications need to register for health monitor through “Application Service.” Application Service monitors the health status of all the mandatory applications and feeds the information to teaming. If any of the application is unhealthy, node is pulled out of the team. All the registered applications receive BECOME_SUSPENDED notification. Please refer to “Application Service” section for further detail.

Non mandatory applications are also allowed to join the team using teaming service interfaces. Availability/non-availability of non mandatory applications doesnt make any difference to the teaming state of the node. Like mandatory applications, non-mandatory applications also receive the teaming state change notifications.

Refer to Application Service, for further detail on mandatory & non-mandatory applications.

TeamingService Interface—provides the methods to Join/Leave controller team, trigger election, read the healthy member information etc., TeamingService runs as an OSGI service. Interested applications need to reference it and consume its services.

Refer to the JavaDocs for TeamingService interface for the detailed explanation of the methods provided and corresponding functionality.

TeamingService Example:

```
import com.hp.sdn.team.TeamingService;
import com.hp.sdn.team.TeamListener;
...
@Component
public class TeamingUser implements TeamListener {
    static final String APP_ID = "TeamingUserApp";
    @Reference(cardinality=ReferenceCardinality.MANDATORY_UNARY,
           policy=ReferencePolicy.DYNAMIC)
    private volatile TeamingService teamingService;

    @Activate
    public void activate() {
        try {
            teamingService.joinTeam(APP_ID, this);
        } catch (Exception e) {
            ...
        }
    }

    @Deactivate
    public void deactivate() {
        try {
            teamingService.leaveTeam(APP_ID, this);
        } catch (Exception e) {
            ...
        }
        ...
    }

    @Override
    public void processTeamEvent(TeamEvent event,
                                 Id<SystemInformation, UUID> instanceId) {
        // Process the message in a separate thread context.
        ...
    }

    private void teamingOperationsExample() {
        // Query for the healthy members of the team
        Collection<String> members = teamingService.getHealthyMembers();

        // Trigger re-election with-in the team
        teamingService.runElection();
    }
}
```

HA Service

HA service provides the building blocks to achieve high availability in the HP VAN SDN Controller environment. This module runs as OSGI service in the Virgo container. Interested applications need to reference HA osgi service. An example java application that makes use of different functionalities of the HA service is described in the subsequent sections.

HA Service is one-stop shop for SDN applications integrating with in memory HA primitives. Through this service applications can make use of the following HA infrastructure blocks:

- Distributed Bus
- Distributed KeyStore
- Distributed Locking

Distributed Bus

In 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. Bus infrastructure 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 the controller cluster nodes.

Distributed Bus provides publish-subscribe system where several applications on different controller nodes can register for various types of bus messages, send and receives messages without worrying about delivery failures or out of order delivery. When an application pushes a message, all the subscribers to that message type are notified through Bus irrespective of their location in the controller cluster.

Distributed Bus provides support for 3 types of messages:

- NO_ACK message
- ONE_ACK messages
- ALL_ACK messages

With NO_ACK type of messages, message post is successful as soon as the message is written onto the bus, not caring about listeners receiving the message and processing it. With ONE_ACK and ALL_ACK type of messages, registered listeners of the message are expected to acknowledge the receipt of the message.

While posting ONE_ACK or ALL_ACK message, application need to register a notification handler for the message delivery status processing. ONE_ACK message status listener is invoked with success response if at least 1 listener responds to the message receipt. With ALL_ACK message all the registered listeners are expected to ACK. Message generating application is notified of the ACK status along with the list of listeners ACKed/NON-ACKed. Listener signifies an application instance that registered for the specific ONE_ACK/ALL_ACK message type. In case of ONE_ACK and ALL_ACK messages, If the desired number of listeners do not ack, timeout occurs and failure is notified to the sender along with the listener list.

Applications can make use of the methods exposed by the following interfaces to work with Distributed Bus.

HAService extends BusFactory interface and hence methods of this interface shall be accessed via HAService in the HP VAN SDN Controller environment.

BusFactory Interface—acts as a book keeper for various types of bus types. A bus type can be created or queried through the methods of this interface.

Bus Interface—provides the methods to operate on a specific bus type. Using this interface methods an application can perform operations like post a message, read messages, delete a message, register/unregister a listener, acknowledge a ONE_ACK/ALL_ACK message etc.,

BusListener Interface—needs to be implemented by the bus message listener application. While registering for bus messages, application needs to pass the object that implements BusListener interface. On seeing a bus message of application interest, method of the registered listener busListener.processBusMessage (BusMessage) is invoked by Bus infrastructure. The order of notification depends on the priority level of the listener (as per BusListener.getListenerPriority()). Listener with lower priority value gets notified first.

BusMessage—represents a bus message. Applications need to create a BusMessage and past it to the interested applications via Bus Interface. Listener application can access various parameters of a message via BusMessage object methods.

Please refer to the JavaDocs for BusFactory, Bus, BusListener interfaces for the detailed explanation of the methods provided and corresponding functionality.

Bus Service Example:

```
import com.hp.dist.bus.Bus;
import com.hp.dist.bus.BusListenerId;
import com.hp.dist.bus.BusMessage;
import com.hp.sdn.ha.HAService;
...
@Component
public class BusUser extends BusListener {
    private Bus bus = null;
    private  BusListenerId id = new BusListenerId("AppName", "Inst1");
    @Reference(cardinality=ReferenceCardinality.MANDATORY_UNARY,
                        policy=ReferencePolicy.DYNAMIC)
    private volatile HAService haService;

    @Activate
    public void activate(String instanceId) {
        try {
            bus = haService .getBus(BusUserBusTypes.class);
        } catch (Exception e) {
            ...
        }
        Set <Integer> enumList = new HashSet<Integer>();
        for (BusUserBusTypes e : EnumSet.allOf(BusUserBusTypes.class)) {
              enumList.add(e.ordinal());
        }
        bus.registerListener(enumList, this);
    }

    @Deactivate
    public void deactivate() {
        bus.unregisterListener(id);
    }

    private void busMessagePostSample() {
        BusMessage msg = new BusMessage(
            BusUserBusType.BUS_USER_MSGTYPE_1, new byte[] {'0', '1'});
        try {
            bus.post(msg);
        } catch (Exception e) {
            ...
        }
    }

    private void sendAckForMessage(BusMessage msg) {
        try {
            bus.ackMessage(msg.getMessageId(), this);
        } catch (Exception e) {
            ...
        }
    }

    @Override
    public BusListenerId getInstanceId() {
        return id;
    }

    @Override
    public int getListenerPriority() {
        return 10;
    }

    @Override
    public void processBusMsg(final BusMessage msg) {
        Runnable msgProcessor = new Runnable() {
            switch (BusUserBusTypes.fromOrdinal(msg.getMessageType())) {
                case BUS_USER_MSGTYPE_1:
                    ...
                    break;
                case BUS_USER_MSGTYPE_2:
                    ...
                    break;
            }
        };
        threadExecutor.execute(msgProcessor);
    }

    public enum BusUserBusTypes{
        BUS_USER_MSGTYPE_1,
        BUS_USER_MSGTYPE_2,
        ...
    }
}
```

Distributed KeyStore

KeyStore is a distributed <key, Value> store for the HP SDN HA aware applications. Key is in string format and can be choosen by the application. <key, value> written to KeyStore by an application can be read by application instances on same or other node of the controller cluster, provided the accessing controller application instance knows the data access parameters [Application name and key]. Application name separates the key spaces and hence different applications can use same key names in the KeyStore. A user knowing the application name and key can access the key value from any instance of the controller cluster. Applications can perform create, get, update data, delete key operations on a <key, Value> tuples. Using keyStore service, applications can share state with a different application or instance of same application which might exist on different nodes of the controller cluster.

A value read is always local and it might not be the latest value [in case the value is updated in another node context and it is yet to sync to the local node]. If the application desires to fetch the latest value it can make use of Get latest value interface which synchronizes the value across nodes and returns the value. Application can make use of locking services to protect a <Key, Value> when there are parallel access [reads, writes] to the value, in which case it suggested to use getLatestValue on the KeyValueNode

Applications can make use of the methods exposed by the following interfaces to work with KeyStore.

HAService extends KeyStore interface and hence methods of this interface shall be accessed via HAService in HP SDN environment.

KeyStore interface—provides the methods to store <Key, Value> tuples and query for them in distributed KeyStore. A user with knowledge of application name and Key string can fetch the value from any instance of the controller cluster.

KeyValueNode interface—provides the method to operate on a single <Key, Value> tuple. KeyStore create or query interface methods return KeyValueNode. Using this interface methods several operations like set value, get value, get latest value, get creator application name, get key string etc., on a single <Key, Value> tuple.

Please refer to the JavaDocs for KeyStore and KeyValueNode interfaces for the detailed explanation of the methods provided and corresponding functionality.

KeyStore Service Example:

```
import com.hp.sdn.ha.HAService;
import com.hp.dist.keystore.KeyStore;
import com.hp.dist.keystore.KeyValueNode;
...
@Component
public class KeyStoreUser {
    private static String appName = "DemoApp";
    @Reference(cardinality=ReferenceCardinality.MANDATORY_UNARY,
        policy=ReferencePolicy.DYNAMIC)
    private volatile HAService haService;

    private void keyStoreOperations() {
        KeyValueNode node = null;
        try {
            node = haService.createNode(appName, "Key",
                                           new byte[]{'d', 'a', 't', 'a'});
        } catch (Exception e) {
            ...
        }

        try {
            node.setValue(
                new byte[] {'n', 'e', 'w', 'd', 'a', 't', 'a'}, true);
        } catch (Exception e) {
            ...
        }

        try {
            node.delete(true);
        } catch (Exception e) {
            ...
        }
    }

    private void keyQueryOperations() {
        KeyValueNode node = null;
        try {
            node == haService.getNode(appName, "Key");
        } catch (KeyNotFoundException e) {
            ...
        } catch (Exception e) {
            …
        }

        List<String> keyList = null;
        try {
            keyList = haService.getKeys(appName);
        } catch (Exception e) {
            ...
        }
    }
}
```

Distributed Locking

Protecting the access to shared resources becomes increasingly important in the distributed environment. Distributed Locking component of HA service helps in achieving serialized access to shared resource.

Application needs to define a key that can be used to protect parallel access to a shared resource. Applications on different controller nodes should agree upon the key and acquire necessary lock on it before accessing the shared resource.

Following are the types of locks supported by distributed locking service:

- **Read Lock** is a shared lock. More than one read lock can be allowed on a key simultaneously.
- **Write Lock** is an exclusive lock. No previous lock [read or write] can be in held state before a write lock can be granted. Read lock cannot be granted until all the previous write locks are released.
- **Multi Lock** is a pool of read and/or write locks. A multi lock is granted only if all the member locks are acquired.

While creating a lock, application needs to provide application Id [app name and instance id] and a lock hierarchy [key] which uniquely identifies the lock.

In case of an application instance failure, its peer application instance running on a different controller instance can take over the locks created by the failed application. Newer application needs to query for the locks created by the failed instance using application name and instance Id. To take over locks of an application instance, the new application instance needs to query lock service for the locks owned by the older application. Based on the need, ownership of the locks can be changed using the lock service methods.

HAService extends LockService interface and hence methods of this interface shall be accessed via HAService in HP SDN environment.

LockService interface—provides the methods for instantiating different type of locks and query them during fail over scenarios. An existing lock can be deleted through this interface methods.

Lock Interface—gives application access to a Lock. Enables applications to perform specific operations like acquire, tryAcquire, unlock on a Lock.

Please refer to the JavaDocs for LockService and Lock interfaces for the detailed explanation of the methods provided and corresponding functionality.

Lock Service Example:

```
import com.hp.sdn.ha.HAService;
import com.hp.dist.lock.LockService;
import com.hp.dist.lock.Lock;
...
@Component
public class LockUser {
    private static String appName = "LockUserApp";
    private static ApplicationId id = new ApplicationId(appName, "Inst1");
    @Reference(cardinality=ReferenceCardinality.MANDATORY_UNARY,
        policy=ReferencePolicy.DYNAMIC)
    private volatile HAService haService;

    private void readLockOperations() {
        Lock readLock = null;
        LockName name = new LockName(appName, "Key1");
        try {
            readLock = haService.createReadLock(id, name);
        } catch (Exception e) {
            ...
        }

        try {
            boolean lockAcquired = readLock.tryLock();
            if (true == lockAcquired)
                readLock.unlock();
            ...
            readLock.lock();
            readLock.unlock();
            ...
            haService.deleteLock(readLock);
        } catch (Exception e) {
            ...
        }
    }

    private void writeLockOperations() {
        Lock writeLock = null;
        LockName name = new LockName(appName, "Key1");
        try {
            writeLock = haService.createWriteLock(id, name);
        } catch (Exception e) {
            ...
        }

        try {
            boolean lockAcquired = writeLock.tryLock(
                10, TimeUnit.MILLISECONDS);
            if (true == lockAcquired)
                writeLock.unlock();
            ...
            writeLock.lock();
            writeLock.unlock();
            ...
            haService.deleteLock(writeLock);
        } catch (Exception e) {
            ...
        }
    }

    private void multiLockOperations() {
        Lock multiLock = null;
```

```
            List<LockNameType> lockNames = new ArrayList<LockNameType>();

            lockNames.add(new LockNameType(LockType.READ_LOCK, appName, "KEY1"));
            lockNames.add(new LockNameType(LockType.WRITE_LOCK,
                appName, "KEY3"));
            // MultiLock name should be unique. While constructing the name,
            // its suggested to prefix application instance id
            multiLock = haService.createMultiLock(
                id, new LockName(appName, "MultiLock"), lockNames);

            try {
                boolean lockAcquired = multiLock.tryLock(
                    10, TimeUnit.MILLISECONDS);
                if (true == lockAcquired)
                    multiLock.unlock();
                ...
                multiLock.lock();
                multiLock.unlock();
                ...
                haService.deleteLock(multiLock);
            } catch (Exception e) {
                ...
            }
        }

        private void lockQueryOperations() {
            try {
                List<LockName> lockNameList = new ArrayList<LockName>();
                lockNameList.add(new LockName(appName));
                List<Lock> locks = haService.getAppLocks(lockNameList, id);
            } catch (Exception e) {
                ...
            }
        }
    }
```

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 leverging 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 sitations:

1) Controller failure: The ROS detects the failure of a controller in a team via notifications from the teaming subsystem. If the ROS determine 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 instane 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 slavs 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 fo 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:

```
import com.hp.sdn.adm.role.RoleService;
import com.hp.sdn.adm.system.SystemInforamationService;
…
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 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 privilages
               // we can trigger write-operations's from that controller.
               ...
               break;
            case SLAVE:
               // we have only read privilages
               ...
               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:

```
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:

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

Persistence

Distributed Persistence Overview

The SDN Controller provides a distributed persistence for applications in form of a Cassandra [11] 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 application 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 [31].

Figure 33 Data Access Object Pattern

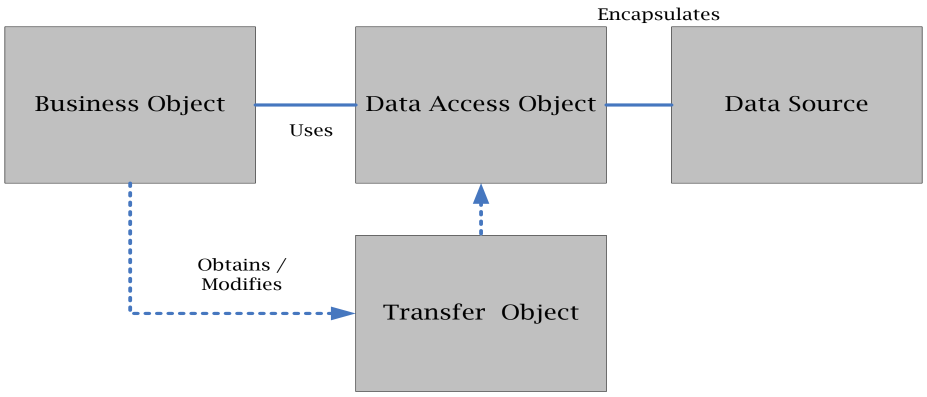
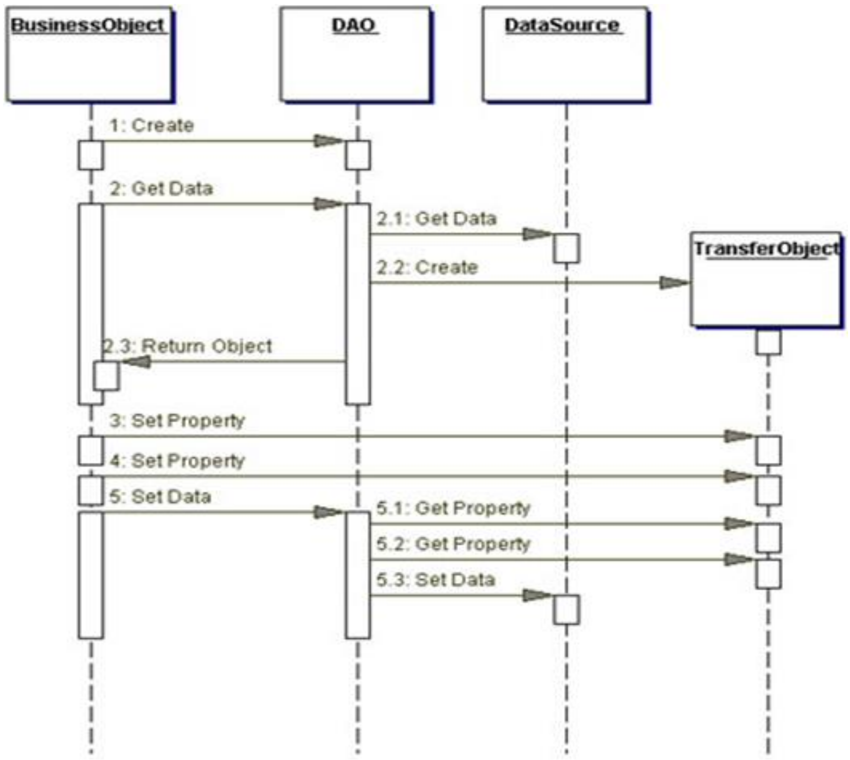


Figure 34 DAO pattern



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

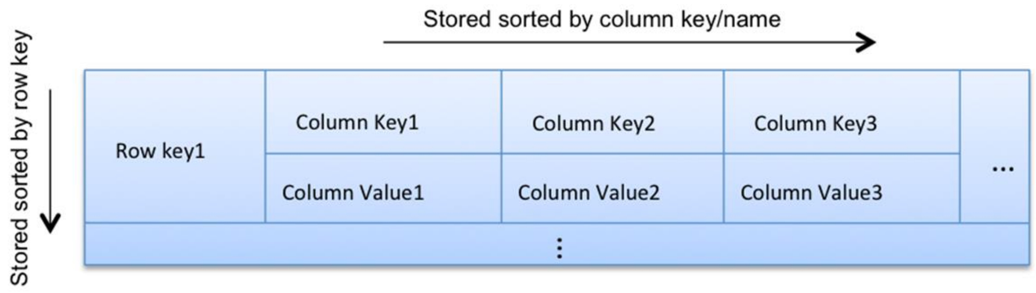
Relational Model	Cassandra Model
Database	Keyspace
Table	Column Family (CF)
Primary key	Row key
Column name	Column name/key
Column value	Column value

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:

```
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 Casssandra row.

Figure 36 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 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 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, I have created a Demo application that persists Alerts in the Distributed Database (Cassandra).

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 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 one by using an OSGi @Reference as shown below.

CassandraAlertManager.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);

    ReadQuery<CassandraAlert, C> getFindAlertByUidAndSysIdQuery(
        String uid, String sysId);

    WriteQuery<CassandraAlert, C> getUpdateAlertStateQuery(
        CassandraAlert alert);

    WriteQuery<Long, C> getTrimAlertQuery(CassandraAlertFilter alertFilter);

    WriteQuery<Long, C> getAddAlertListQuery(List<CassandraAlert> alerts);

    WriteQuery<Long, C> getUpdateAlertListQuery(
        List<String> uids, String sysId, boolean state);

    WriteQuery<Long, C> getDeleteAlertListQuery(
        List<String> uids, String sysId);

    ReadQuery<Long, C> getCountAlertQuery();
}
```

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:

```
@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) {
        ...
    }
    return alert;
}
```

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 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:

```
@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) {
        ...
    }
}
```

```
@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 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:

```
package com.hp.hm.model;

import com.hp.util.filter.EqualityCondition;
import com.hp.util.filter.SetCondition;
import com.hp.util.filter.StringCondition;
...
public class CasssandraAlertFilter {

    private SetCondition<Severity> severityCondition;
    private EqualityCondition<Boolean> stateCondition;
    private StringCondition topicCondition;
    private StringCondition originCondition;
    ...
    // Implement setters and getters for all conditions.
    // Good practice to override toString()
}
```

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:

```
package com.hp.demo.cassandra.dao.impl;
...
public class CassandraAlertDao extends
    CassAbstractDao<String, String, CassandraAlert,
        CassandraStorable<String, String>, CassandraAlertFilter,
             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 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;
                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:

```
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);
    }

    @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.

convert()

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:

```
@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);
    }

    return alert;
}
```

getColumnDecoder()

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

CassandraAlertDao.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:

```
@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.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;
}
```

getColumnFamilyDefinitions()

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

CassandraAlertDao.java:

```
    @Override
    protected Collection<ColumnFamilyDefinition<?, ?>>
                                         getColumnFamilyDefinitions() {
        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));
                }
            };

            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);
                }
            };
    // Start the transaction
    context.getTransactionContext()
           .prepareTransaction(AlertsByTopic.COL_FAMILY.getName(),
                               AlertsByTopic.ROW_KEY);

    CassandraStorable<String, Topic> rows = context
            .getTransactionContext()
            .executeCriticalSection(procedure);
    // Add the rows to the row set
    Collection<String> id = new ArrayList<String>();
    for (Column<Topic, ?> topic : rows.getColumns()) {
        id.add(topic.getName().getValue().getId());
    }

    // Add row Id's to the final Id set
    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:

```
@Override
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");
        }

        // 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()
                    .getValue());
            return new MarkPage<String>(pageRequest1, id);
        }
        return null;
    }
```

compareRows()

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

CassandraAlertDao.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.

4 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 automatizes the steps described in the following information. Thus, if preferred (and it is recommended) using the application generator tool to create an application workspace go directly to Application Generator (Automatic Workspace Creation) on page 114.

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.

It is assumed the pre-requisites are already met; see 2 Setting Environment on page 7.

Application Description

For this example 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 2 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 2.

Table 2 Sample Application Information

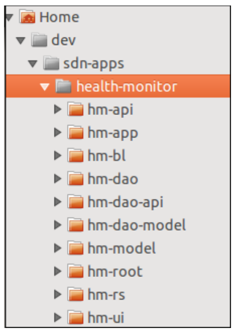
Property	Value
Application Name	Health Monitor
Application Short Name	hm
Company	Hewlett-Packard
Company Short Name	Hp

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 3 describes the folders under the main application directory (*health-monitor* for this sample application). Table 3Figure 38 shows the application module dependencies.

Figure 37 Application Directory Structure



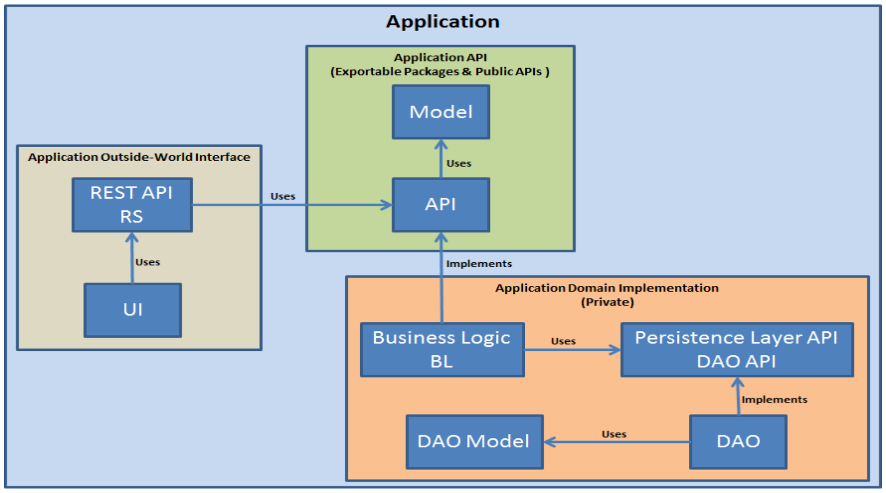
```
Home
  dev
    sdn-apps
      health-monitor
        hm-api
        hm-app
        hm-bl
        hm-dao
        hm-dao-api
        hm-dao-model
        hm-model
        hm-root
        hm-rs
        hm-ui
```

Table 3 Application Folders

Folder	Type	Description
hm-app	Configuration Folder	It contains the application deployment OSGi plan, the application descriptor and a POM file used to generate the installable application.
hm-root	Configuration Folder	Keeps the parent pom.xml file which contains common properties and dependencies to all modules.
hm-model	Module / Source Code Project	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.
hm-api	Module / Source Code Project	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.

hm-bl	Module / Source Code Project	Business logic. Implementation of the application's API. This project is private to the application.
hm-rs	Module / Source Code Project	Application's Representational State Transfer (REST) API or RESTful Web Services. This project is private to the application; however RESTful web services are accessible via HTTP protocol.
hm-ui	Module / Source Code Project	Application's WEB interface. This project is private to the application.
hm-dao-api	Module / Source Code Project	Defines the persistence layer API. This API will be available to the Business Logic to perform database operations. This project is private to the application.
hm-dao-model	Module / Source Code Project	Defines the persistent entities. This project is private to the application.
hm-dao	Module / Source Code Project	Persistence layer implementation. This project is private to the application.

Figure 38 Application Project Dependencies



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 2.

Sample Application Root POM File:

```
<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-dao-api</module>
        <module>../hm-dao-model</module>
        <module>../hm-dao</module>
        <module>../hm-bl</module>
        <module>../hm-rs</module>
        <module>../hm-ui</module>
        <module>../hm-app</module>
    </modules>

    <properties>
        <hp-util.version>6.23.0</hp-util.version>
        <sdn.version>2.0.0</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 3 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 2. A `pom.xml` for each application module must be created under the module's folder.

Sample Application Module POM File:

```
<?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 Deployment Plan

Plans encapsulate the artifacts of an application as a single unit. Plans are XML files that have a .plan file extension, such as *multi-artifact.plan*. The structure of the XML file is simple: the root element is <plan> with attributes specifying the name of the plan, the version, atomicity, and scoping. Then, for each artifact that makes up the application, a <artifact> element is added, using its attributes to specify the type of artifact and its name and version. A template for the plan can be found in the HP VAN SDN Controller SDK.

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

Sample Application Deployment Plan:

```
<?xml version="1.0" encoding="UTF-8"?>
<plan name="Health Monitor" 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-dao.api"
version="1.0.0.SNAPSHOT "/>
    <artifact type="bundle" name="com.hp.hm.hm-dao.model"
version="1.0.0.SNAPSHOT "/>
    <artifact type="bundle" name="com.hp.hm.hm-dao" version="1.0.0.SNAPSHOT
"/>
    <artifact type="bundle" name="com.hp.hm.hm-bl" version="1.0.0.SNAPSHOT
"/>
</plan>
```

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:

```
App_id=com.hp.hm
App_name=Health-Monitor
Version=1.0.0.SNAPSHOT
Support_team=false
Vendor= Hewlett-Packard
Description=Health Monitor SDN Application
```

Any value can be set to the *App_id* property as long as it is unique across installed applications. In the same way any value can be set to the *App_name* property; however consider that this value is the one that appears in the SDN controller's Applications view. *Version* attribute must match the application's version configured in the POM files.

Support_team specifies whether the application is mandatory from the team point of view. This attribute has implications on the way controllers handle a failure. If a mandatory application fails, the controller declares itself as non-operational and the team will react upon such a failure. If this attribute is set as *false* the application can still be part of a team, however the controller is still operational in the event of a failure in the application.

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), plus the application deployment plan file *("…/health-monitor/hm-app/hm.plan"*).

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 114. 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:

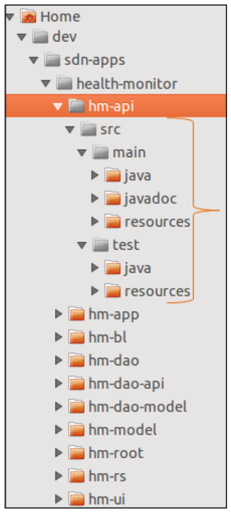
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">

    <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>
        <!— Add a dependency for each app module except rs and ui -->
        <dependency>
            <groupId>com.hp.hm</groupId>
            <artifactId>hm-model</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/">
```

```
                                        <!— Add an <include> node for api,
bl, dao-api, dao-model and dao -->
                                        <include name="hm-
model/${project.version}/hm-model-${project.version}.jar"/>
                                    </fileset>
                                    <fileset dir="${basedir}"
includes="hm.plan"/>
                                    <fileset dir="${basedir}"
includes="hm.descriptor"/>
                                </copy>
                                <zip destfile="target/hm-
${project.version}.zip" basedir="target/bundles"/>
                            </tasks>
                        </configuration>
                        <goals>
                            <goal>run</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

Creating Module Directory Structure

At this point there should be a pom.xml file under each folder listed in Table 3. 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 4.

Table 4 Sample Application Generator Parameters

Parameter	Description
target	Target where the application code is to be generated
app	One word application name in lower-case, for example, 'hm'
company	One word company name in lower-case, for example, 'hp'
subject	One word short subject in camel-case, for example, 'OpenFlowSwitch'
app-name	Optional full application name, for example, 'Health Monitor'
company-name	Optional full company name, for example, 'Hewlett-Packard'
app-description	Optional brief description of the application
rest-path	Optional REST API path , e.g. 'health'

To protect any existing code customizations, the *'target'* 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 we use the application generator.

```
$ bin/gen-sdn-app ~/dev/sdn-apps/health-monitor hm hp OpenFlowSwitch \
    "Device Health Monitor" "Hewlett-Packard" \
    "Application for monitoring health of network devices." health
```

When executing the command with no parameters the command’s documentation displays, 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 above.

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 Application on page 116, 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 elipse:eclipse
```

Once the maven command completes, the projects can be imported from within the Eclipse IDE, see Importing Java Projects on page 208.

Updating Project Dependencies

The command described in Creating Eclipse Projects above creates the Eclipse projects resolving all dependencies desfined 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 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 216 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 Application on page 116 to directly upload the application zip file.In order to property 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 114.

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 5 lists suggestions.

Application Module Seed Java Class:

```
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 5 Suggested Java Packages

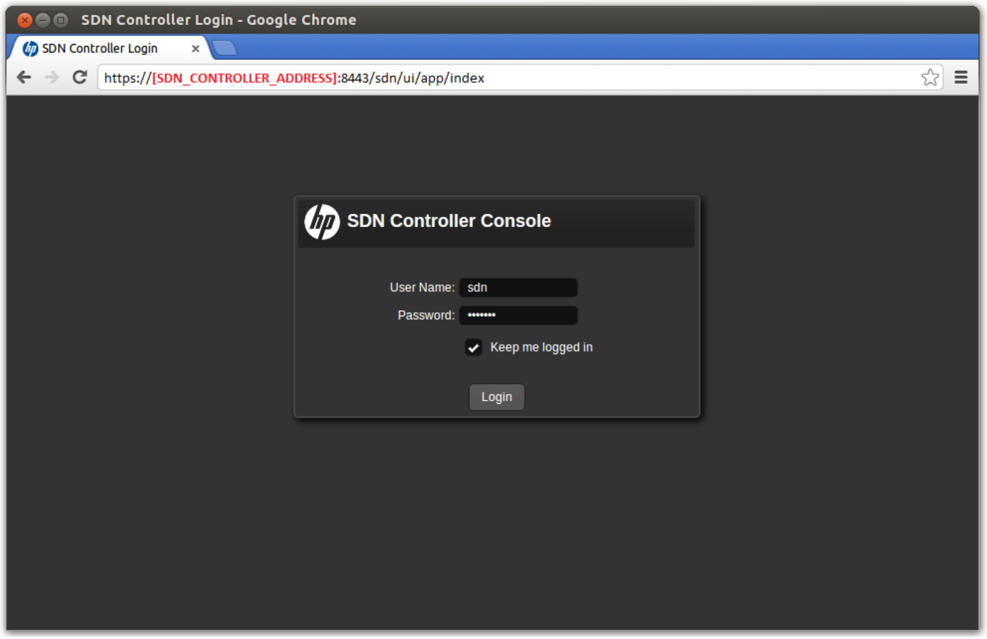
Module	Recommended Package	Example
Model	com.[COMPANY_SHORT_NAME].[APPLICATION_SHORT_NAME].model	com.hp.hm.model
API	com.[COMPANY_SHORT_NAME].[APPLICATION_SHORT_NAME].api	com.hp.hm.api
Business Logic	com.[COMPANY_SHORT_NAME].[APPLICATION_SHORT_NAME].impl	com.hp.hm.impl
REST API	com.[COMPANY_SHORT_NAME].[APPLICATION_SHORT_NAME].rs	com.hp.hm.rs
UI	com.[COMPANY_SHORT_NAME].[APPLICATION_SHORT_NAME].ui	com.hp.hm.ui
DAO API	com.[COMPANY_SHORT_NAME].[APPLICATION_SHORT_NAME].dao.api	com.hp.hm.dao.api
DAO Model	com.[COMPANY_SHORT_NAME].[APPLICATION_SHORT_NAME].dao.model	com.hp.hm.dao.model
DAO	com.[COMPANY_SHORT_NAME].[APPLICATION_SHORT_NAME].dao.impl	com.hp.hm.dao.impl

Installing 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.

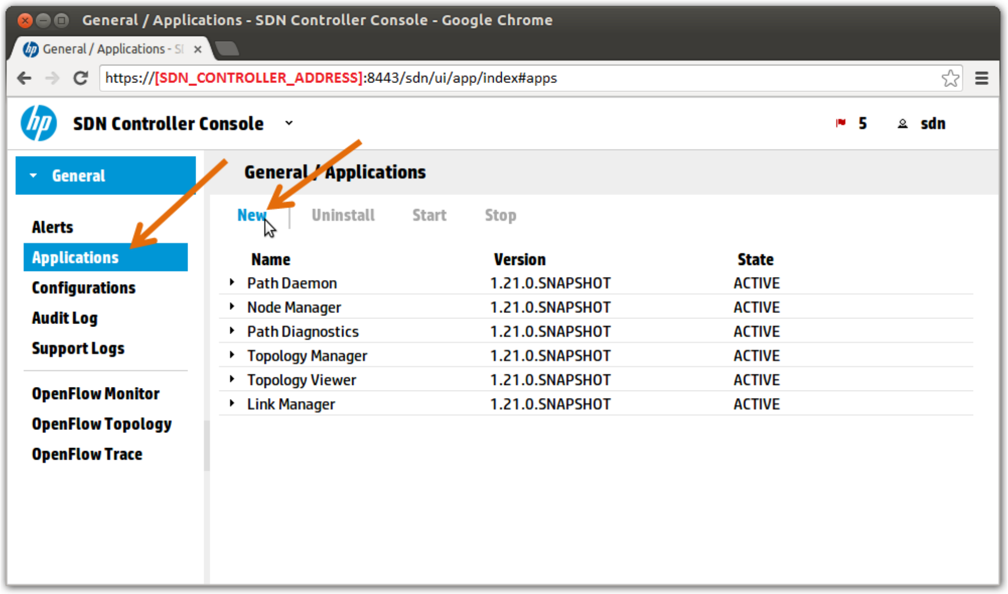
1. Login to the SDN controller as ***sdn*** user using the following URL *"https://**SDN_CONTROLLER_ADDRESS]**:8443/sdn/ui/"* as shown in Figure 40. See Authentication Configuration on page 7 to determine the right credentials.

Figure 40 SDN Controller Login Page



2. Click the ***New*** tool bar action as illustrated in Figure 41.

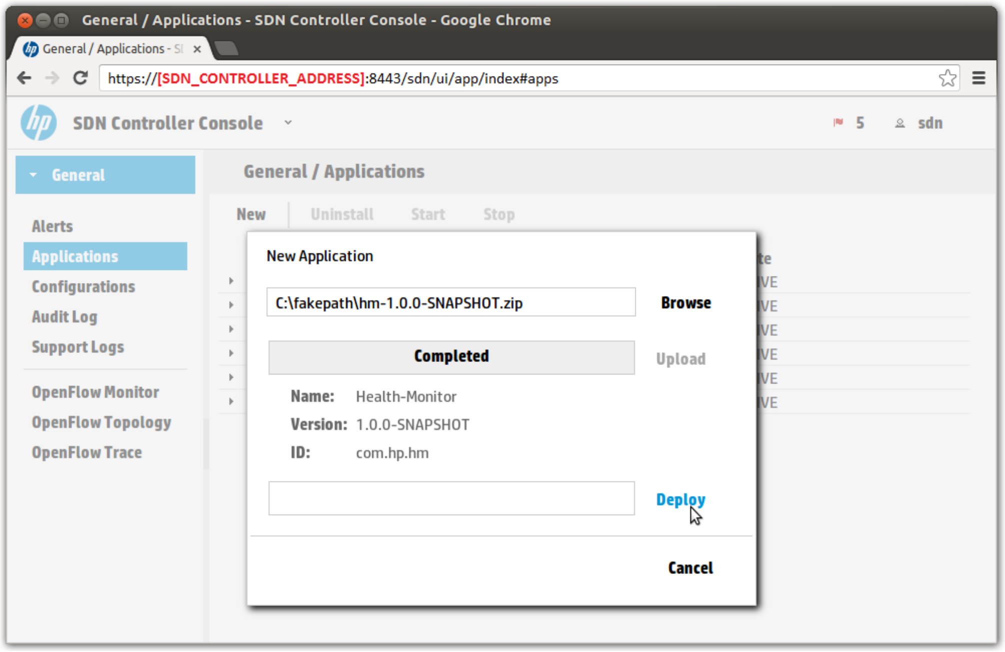
Figure 41 SDN Controller Applications



Name	Version	State
Path Daemon	1.21.0.SNAPSHOT	ACTIVE
Node Manager	1.21.0.SNAPSHOT	ACTIVE
Path Diagnostics	1.21.0.SNAPSHOT	ACTIVE
Topology Manager	1.21.0.SNAPSHOT	ACTIVE
Topology Viewer	1.21.0.SNAPSHOT	ACTIVE
Link Manager	1.21.0.SNAPSHOT	ACTIVE

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 111.
5. Click *Deploy* to install the application.

Figure 42 Upload/Install Application

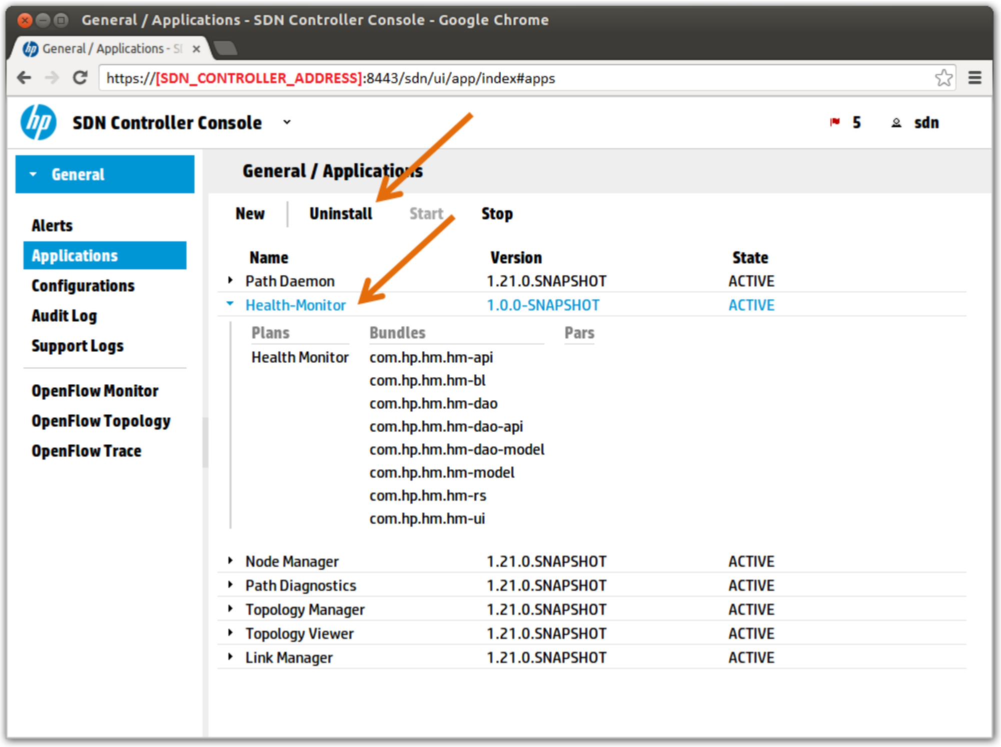


C:\fakepath\hm-1.0.0-SNAPSHOT.zip

Name: Health-Monitor
Version: 1.0.0-SNAPSHOT
ID: com.hp.hm

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.

Figure 43 Application Management



Name	Version	State
Path Daemon	1.21.0.SNAPSHOT	ACTIVE
Health-Monitor	1.0.0-SNAPSHOT	ACTIVE
Node Manager	1.21.0.SNAPSHOT	ACTIVE
Path Diagnostics	1.21.0.SNAPSHOT	ACTIVE
Topology Manager	1.21.0.SNAPSHOT	ACTIVE
Topology Viewer	1.21.0.SNAPSHOT	ACTIVE
Link Manager	1.21.0.SNAPSHOT	ACTIVE

Plans	Bundles	Pars
Health Monitor	com.hp.hm.hm-api	
	com.hp.hm.hm-bl	
	com.hp.hm.hm-dao	
	com.hp.hm.hm-dao-api	
	com.hp.hm.hm-dao-model	
	com.hp.hm.hm-model	
	com.hp.hm.hm-rs	
	com.hp.hm.hm-ui	

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 importat difference with previous illustrations of the same code. You can get the complete sample application source code from the HP VAN SDN Controller SDK [19].

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 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 [32]. This example uses a S*witch* 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 space limitations illustrating code samples).

Switch.java:

```
package com.hp.hm.model;

import com.hp.api.Id;
import com.hp.sdn.AbstractModel;
import com.hp.util.ip.IpAddress;
import com.hp.util.ip.MacAddress;

public class Switch extends AbstractModel<Switch, Long> {
    ...
    private final MacAddress macAddress;
    private IpAddress ipAddress;
    private String friendlyName;
    private ActiveState activeState;
```

```
        public Switch(MacAddress macAddress) {
            this(null, macAddress);
        }

        public Switch(Id<Switch, Long> id, MacAddress macAddress) {
            super(id);
            if (macAddress == null) {
                throw new NullPointerException("macAddress cannot be null");
            }
            this.macAddress = macAddress;
        }

        // Implement getters for immutable fields: MAC Address.
        // Implement setters and getters for mutable fields: IP address,
        // friendly name and active status.

        // Good practice to override the following methods on transport objects:
        // equals(Object), hashCode() and toString()
        ...
    }
```

ActiveState.java:

```
    package com.hp.hm.model;

    public enum ActiveState {
        ON,
        OFF
    }
```

The *Switch* model object implements the *Transportable* interface which is part of the HP VAN SDN Controller Framework; *AbstractModel* offers a partial implementation of such 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* and *Transportable* are used just to denote that *Switch* follows the data transfer object pattern [32]. *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 persiste data, then data transfer objects take an explicit role and they must follow certan hierarchically 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 Persistence on page 79 to choose the right interface if data will be persisted.

Open the *hm-root/pom.xml* file and add the XML extract from following list to the *<dependencies>* node. After updating the POM file update the Eclipse project dependencies (see Updating Project Dependencies on page 115).

HP SDN Controller Framework Common Dependencies:

```
<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>
```

Switch model object implements *Transportable<Switch, Long>* which means that the *Switch*'s identity (or id) type is Long. Long has been chosen to simplify the implementation, however the services published by the HP VAN SDN Controller normally use *java.util.UUID*.

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 [20] 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:

```
package com.hp.hm.model;

import com.hp.util.filter.EqualityCondition;
import com.hp.util.filter.SetCondition;
import com.hp.util.filter.StringCondition;
...
public class SwitchFilter {

    private EqualityCondition<MacAddress> macAddressCondition;
    private SetCondition<IpAddress> ipAddressCondition;
    private StringCondition friendlyNameCondition;
    private EqualityCondition<ActiveState> activeStateCondition;
    ...
    // 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 in non-active state with a friendly name that contains the text 'My Switch' and an IP Address that is not one of the following: [192.168.1.2, 192.168.1.3].*

Switch Filter Usage Example:

```
SwitchFilter filter = new SwitchFilter();
filter.setActiveStateCondition(new
EqualityCondition<ActiveState>(ActiveState.OFF,
EqualityCondition.Mode.EQUAL));
filter.setFriendlyNameCondition(new StringCondition("My Switch",
StringCondition.Mode.CONTAINS));

Set<IpAddress> ips = new HashSet<IpAddress>();
ips.add(IpAddress.valueOf("192.168.1.2"));
ips.add(IpAddress.valueOf("192.168.1.3"));
filter.setIpAddressCondition(new SetCondition<IpAddress>(ips,
SetCondition.Mode.NOT_IN));
```

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 Mac Address ascending, then Active State descending and lastly Friendly Name ascending.*

SwitchSortKey.java:

```
package com.hp.hm.model;

public enum SwitchSortKey {
    MAC_ADDRESS,
    FRIENDLY_NAME,
    ACTIVE_STATE
}
```

Switch Sort Specification Usage Example:

```
SortSpecification<SwitchSortKey> sort =
    new SortSpecification<SwitchSortKey>();
sort.addSortComponent(SwitchSortKey.MAC_ADDRESS, SortOrder.ASCENDING);
sort.addSortComponent(SwitchSortKey.ACTIVE_STATE, SortOrder.DESCENDING);
sort.addSortComponent(SwitchSortKey.FRIENDLY_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 [32] is a pattern while a JavaBean [33] 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:

```
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):

```
package com.hp.hm.api;

import com.hp.api.Id;
import com.hp.hm.model.Switch;
import com.hp.hm.model.SwitchFilter;
import com.hp.hm.model.SwitchSortKey;
import com.hp.util.SortSpecification;
...
public interface SwitchService {

    public Switch add(Switch device);

    public void update(Switch device);
    public Switch get(Id<Switch, Long> id);

    public List<Switch> find(SwitchFilter filter,
                             SortSpecification<SwitchSortKey> sortSpecification);

    public void delete(Id<Switch, Long> 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 115).

Application Model Dependency:

```
<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 115).

Application API Dependency:

```
<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.

SwitchManager.java (Sample Application Service Implementation):

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

```
...
public class SwitchManager implements SwitchService {

    @Override
    public Switch add(Switch device) {
        return device;
    }

    @Override
    public void update(Switch device) {
    }

    @Override
    public Switch get(Id<Switch, Long> id) {
        return new Switch(id, MacAddress.valueOf("00:00:00:00:00:01"));
    }

    @Override
    public List<Switch> find(SwitchFilter filter,
                        SortSpecification<SwitchSortKey> sortSpecification){
        return Collections.emptyList();
    }

    @Override
    public void delete(Id<Switch, Long> id) {
    }
}
```

Providing Services with OSGi Declarative Services

The OSGi standard component framework, called *Declarative Services* [34], 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 [34].

Continuing with the example, the *SwitchService* from the Service API on page 124 will be published via *SwitchManager* from the Service Implementation on page 125 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 [20]. 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 [35]) 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 is also located in the *hm-bl* module.

NOTE

The usage of *SwitchComponent* may be omitted and directly annotate *SwitchManager* if preferred.

SwitchComponent.java (Sample Application OSGi Service Component):

```
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 add(Switch device) {
            return delegate.add(device);
        }

        @Override
        public void update(Switch device) {
            delegate.update(device);
        }

        @Override
        public Switch get(Id<Switch, Long> id) {
            return delegate.get(id);
        }

        @Override
        public List<Switch> find(SwitchFilter filter,
                                 SortSpecification<SwitchSortKey> sortSpecification) {
            return delegate.find(filter, sortSpecification);
        }

        @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 Application and Installing Application sections.

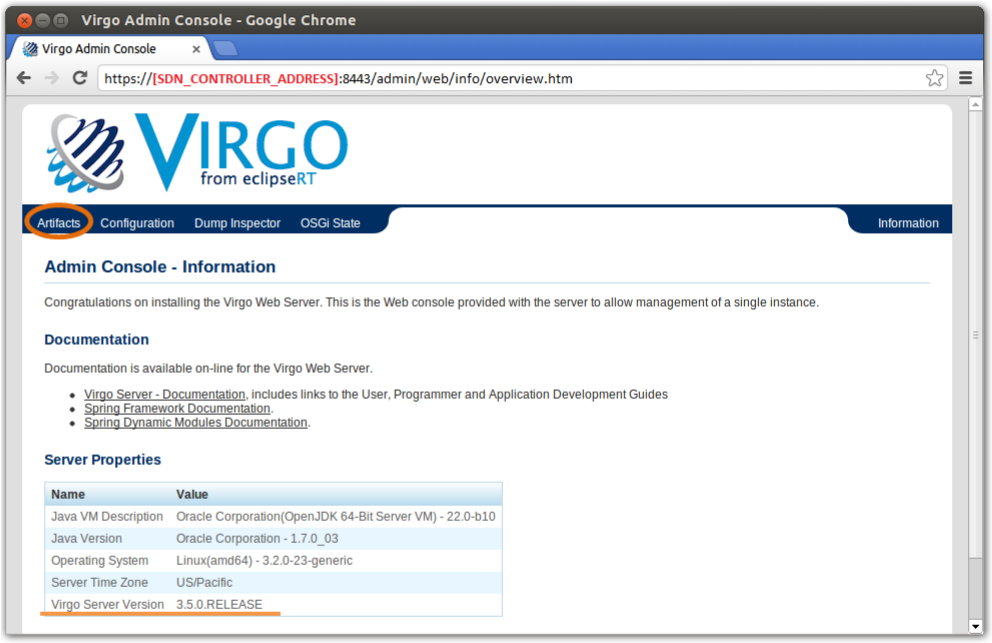
NOTE

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 44, Figure 45, Figure 46 and Figure 47. 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 47.

Figure 44 Virgo 3.5.0 Admin Console


Virgo Admin Console - Google Chrome
Virgo Admin Console
https://[SDN_CONTROLLER_ADDRESS]:8443/admin/web/info/overview.htm
VIRGO
from eclipseRT
Artifacts
Configuration
Dump Inspector
OSGi State
Information

Admin Console - Information

Congratulations on installing the Virgo Web Server. This is the Web console provided with the server to allow management of a single instance.

Documentation

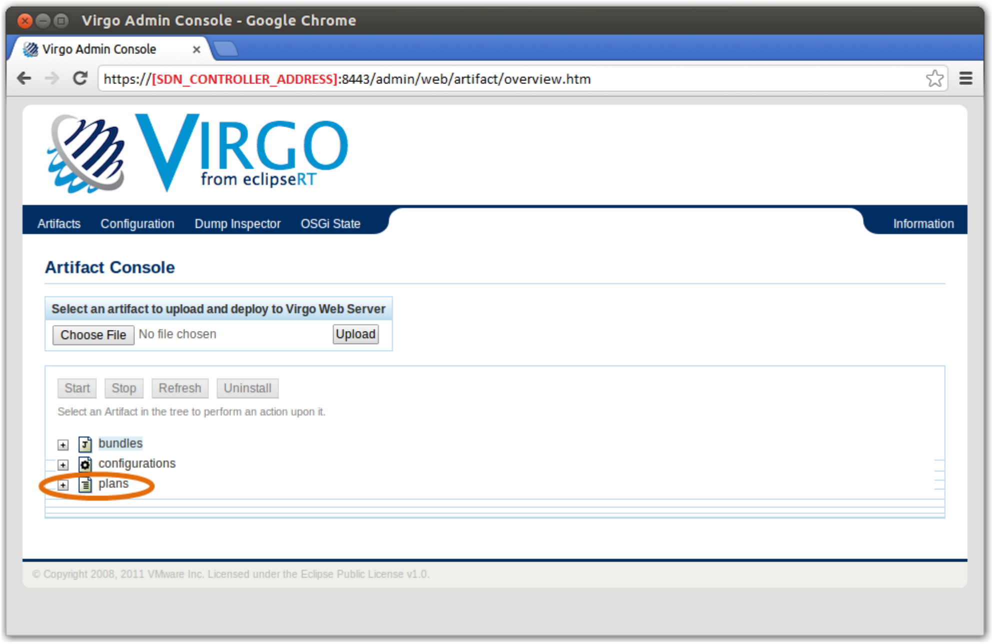
Documentation is available on-line for the Virgo Web Server.

- Virgo Server - Documentation, includes links to the User, Programmer and Application Development Guides
- Spring Framework Documentation.
- Spring Dynamic Modules Documentation.

Server Properties

Name	Value
Java VM Description	Oracle Corporation(OpenJDK 64-Bit Server VM) - 22.0-b10
Java Version	Oracle Corporation - 1.7.0_03
Operating System	Linux(amd64) - 3.2.0-23-generic
Server Time Zone	US/Pacific
Virgo Server Version	3.5.0.RELEASE

Figure 45 Virgo 3.5.0 Admin Console Artifacts



Virgo Admin Console - Google Chrome
Virgo Admin Console
https://[SDN_CONTROLLER_ADDRESS]:8443/admin/web/artifact/overview.htm
VIRGO from eclipseRT
Artifacts
Configuration
Dump Inspector
OSGi State
Information

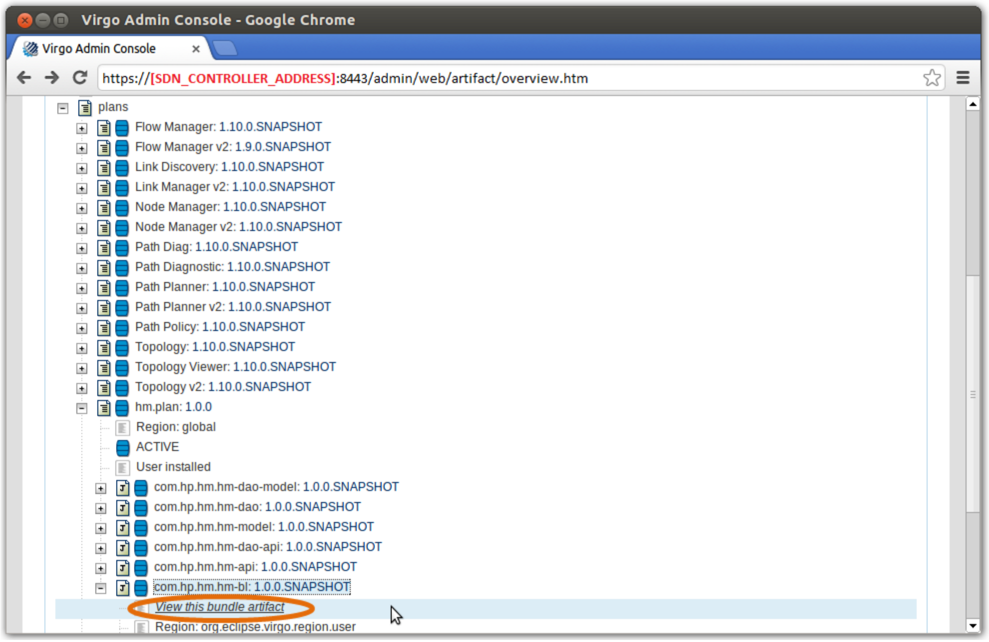
Artifact Console

Select an artifact to upload and deploy to Virgo Web Server

Select an Artifact in the tree to perform an action upon it.

```
+ bundles
+ configurations
+ plans
```

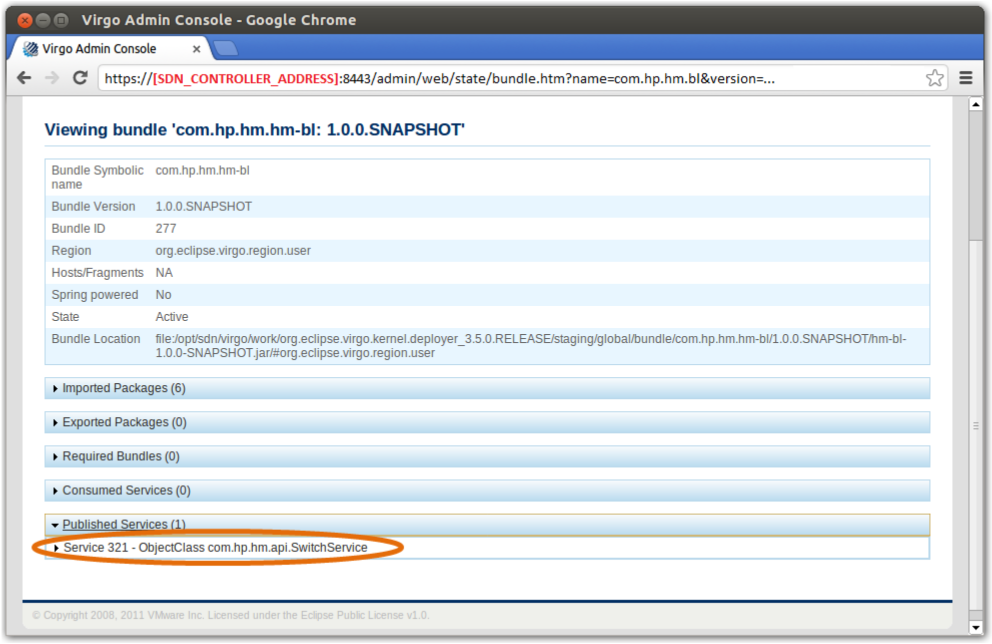
Figure 46 Virgo 3.5.0 Admin Console Application Plan



Virgo Admin Console - Google Chrome
Virgo Admin Console
https://[SDN_CONTROLLER_ADDRESS]:8443/admin/web/artifact/overview.htm

```
− plans
  + Flow Manager: 1.10.0.SNAPSHOT
  + Flow Manager v2: 1.9.0.SNAPSHOT
  + Link Discovery: 1.10.0.SNAPSHOT
  + Link Manager v2: 1.10.0.SNAPSHOT
  + Node Manager: 1.10.0.SNAPSHOT
  + Node Manager v2: 1.10.0.SNAPSHOT
  + Path Diag: 1.10.0.SNAPSHOT
  + Path Diagnostic: 1.10.0.SNAPSHOT
  + Path Planner: 1.10.0.SNAPSHOT
  + Path Planner v2: 1.10.0.SNAPSHOT
  + Path Policy: 1.10.0.SNAPSHOT
  + Topology: 1.10.0.SNAPSHOT
  + Topology Viewer: 1.10.0.SNAPSHOT
  + Topology v2: 1.10.0.SNAPSHOT
  − hm.plan: 1.0.0
      Region: global
      ACTIVE
      User installed
    + com.hp.hm.hm-dao-model: 1.0.0.SNAPSHOT
    + com.hp.hm.hm-dao: 1.0.0.SNAPSHOT
    + com.hp.hm.hm-model: 1.0.0.SNAPSHOT
    + com.hp.hm.hm-dao-api: 1.0.0.SNAPSHOT
    + com.hp.hm.hm-api: 1.0.0.SNAPSHOT
    − com.hp.hm.hm-bl: 1.0.0.SNAPSHOT
        View this bundle artifact
        Region: org.eclipse.virgo.region.user
```

Figure 47 Virgo 3.5.0 Admin Console Application's Business Logic Bundle



Virgo Admin Console - Google Chrome

https://[SDN_CONTROLLER_ADDRESS]:8443/admin/web/state/bundle.htm?name=com.hp.hm.bl&version=...

Viewing bundle 'com.hp.hm.hm-bl: 1.0.0.SNAPSHOT'

Bundle Symbolic name	com.hp.hm.hm-bl
Bundle Version	1.0.0.SNAPSHOT
Bundle ID	277
Region	org.eclipse.virgo.region.user
Hosts/Fragments	NA
Spring powered	No
State	Active
Bundle Location	file:/opt/sdn/virgo/work/org.eclipse.virgo.kernel.deployer_3.5.0.RELEASE/staging/global/bundle/com.hp.hm.hm-bl/1.0.0.SNAPSHOT/hm-bl-1.0.0-SNAPSHOT.jar/#org.eclipse.virgo.region.user

▸ Imported Packages (6)

▸ Exported Packages (0)

▸ Required Bundles (0)

▸ Consumed Services (0)

▾ Published Services (1)

▸ Service 321 - ObjectClass com.hp.hm.api.SwitchService

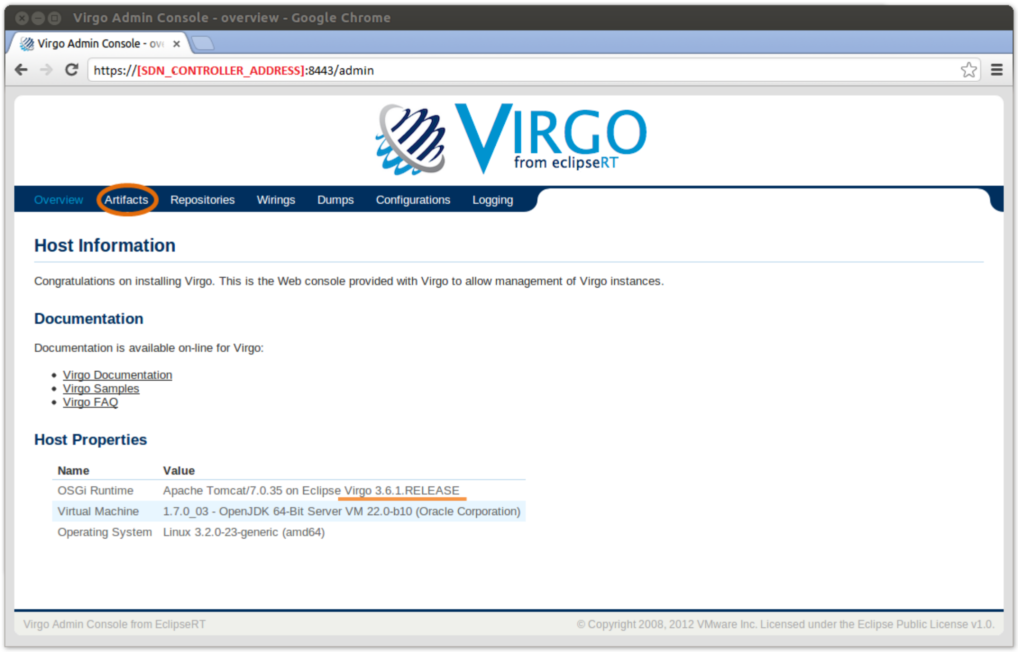
© Copyright 2008, 2011 VMware Inc. Licensed under the Eclipse Public License v1.0.

Virgo 3.6.1

Since Virgo dropped the "Published Services" section illustrated in Figure 47, 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 48, Figure 49, Figure 50, and Figure 51. 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 51.

Figure 48 Virgo 3.6.1 Admin Console


Virgo Admin Console - overview - Google Chrome
https://[SDN_CONTROLLER_ADDRESS]:8443/admin
VIRGO from eclipseRT
Overview
Artifacts
Repositories
Wirings
Dumps
Configurations
Logging

Host Information

Congratulations on installing Virgo. This is the Web console provided with Virgo to allow management of Virgo instances.

Documentation

Documentation is available on-line for Virgo:

- Virgo Documentation
- Virgo Samples
- Virgo FAQ

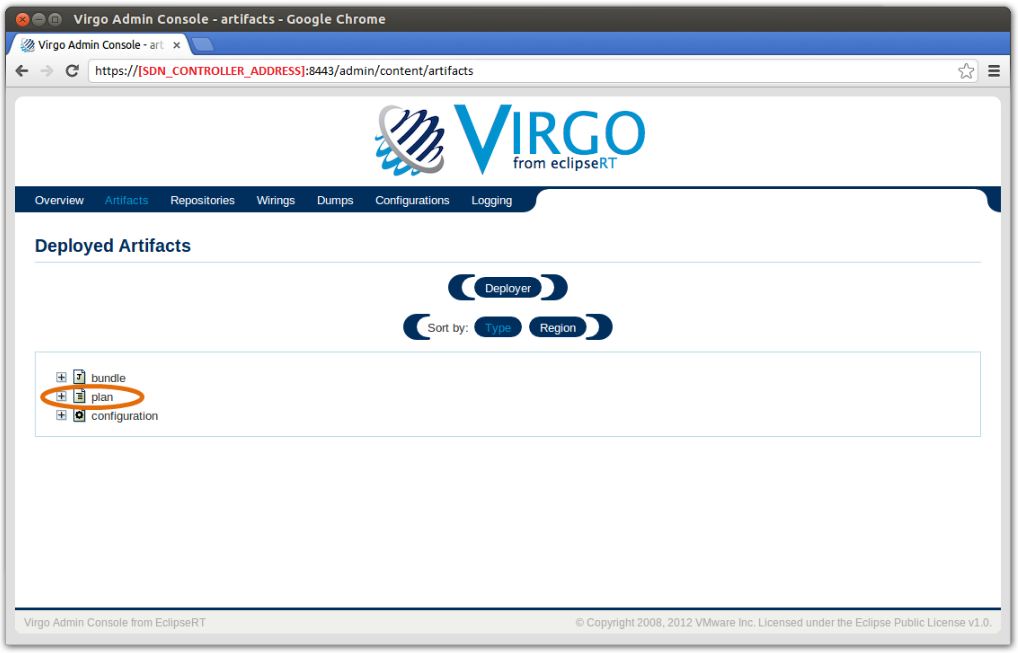
Host Properties

Name	Value
OSGi Runtime	Apache Tomcat/7.0.35 on Eclipse Virgo 3.6.1.RELEASE
Virtual Machine	1.7.0_03 - OpenJDK 64-Bit Server VM 22.0-b10 (Oracle Corporation)
Operating System	Linux 3.2.0-23-generic (amd64)

Virgo Admin Console from EclipseRT

© Copyright 2008, 2012 VMware Inc. Licensed under the Eclipse Public License v1.0.

Figure 49 Virgo 3.6.1 Admin Console Artifacts

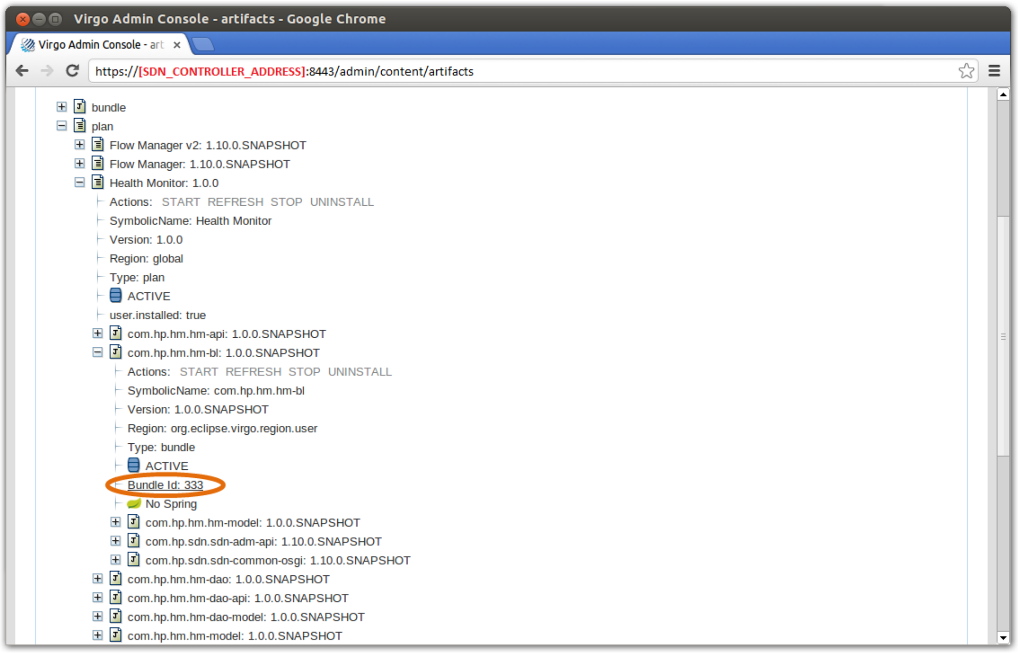


Deployed Artifacts

Virgo Admin Console from EclipseRT

© Copyright 2008, 2012 VMware Inc. Licensed under the Eclipse Public License v1.0.

Figure 50 Virgo 3.6.1 Admin Console Application Plan

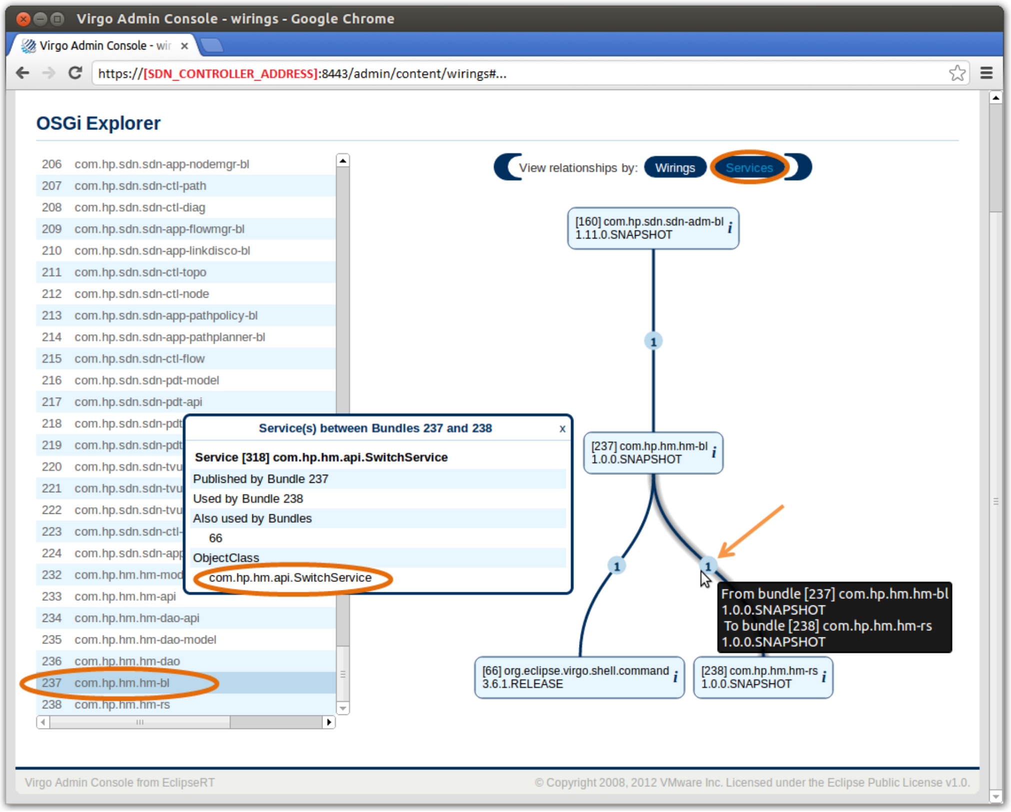


Virgo Admin Console - artifacts - Google Chrome

https://[SDN_CONTROLLER_ADDRESS]:8443/admin/content/artifacts

```
⊞ bundle
⊟ plan
    ⊞ Flow Manager v2: 1.10.0.SNAPSHOT
    ⊞ Flow Manager: 1.10.0.SNAPSHOT
    ⊟ Health Monitor: 1.0.0
        Actions:  START  REFRESH  STOP  UNINSTALL
        SymbolicName: Health Monitor
        Version: 1.0.0
        Region: global
        Type: plan
        ACTIVE
        user.installed: true
        ⊞ com.hp.hm.hm-api: 1.0.0.SNAPSHOT
        ⊟ com.hp.hm.hm-bl: 1.0.0.SNAPSHOT
            Actions:  START  REFRESH  STOP  UNINSTALL
            SymbolicName: com.hp.hm.hm-bl
            Version: 1.0.0.SNAPSHOT
            Region: org.eclipse.virgo.region.user
            Type: bundle
            ACTIVE
            Bundle Id: 333
            No Spring
            ⊞ com.hp.hm.hm-model: 1.0.0.SNAPSHOT
            ⊞ com.hp.sdn.sdn-adm-api: 1.10.0.SNAPSHOT
            ⊞ com.hp.sdn.sdn-common-osgi: 1.10.0.SNAPSHOT
        ⊞ com.hp.hm.hm-dao: 1.0.0.SNAPSHOT
        ⊞ com.hp.hm.hm-dao-api: 1.0.0.SNAPSHOT
        ⊞ com.hp.hm.hm-dao-model: 1.0.0.SNAPSHOT
        ⊞ com.hp.hm.hm-model: 1.0.0.SNAPSHOT
```

Figure 51 Virgo 3.6.1 Admin Console Business Logic Bundle Relationships by Service



Virgo Admin Console - wirings - Google Chrome

https://[SDN_CONTROLLER_ADDRESS]:8443/admin/content/wirings#...

OSGi Explorer

View relationships by: Wirings Services

```
206  com.hp.sdn.sdn-app-nodemgr-bl
207  com.hp.sdn.sdn-ctl-path
208  com.hp.sdn.sdn-ctl-diag
209  com.hp.sdn.sdn-app-flowmgr-bl
210  com.hp.sdn.sdn-app-linkdisco-bl
211  com.hp.sdn.sdn-ctl-topo
212  com.hp.sdn.sdn-ctl-node
213  com.hp.sdn.sdn-app-pathpolicy-bl
214  com.hp.sdn.sdn-app-pathplanner-bl
215  com.hp.sdn.sdn-ctl-flow
216  com.hp.sdn.sdn-pdt-model
217  com.hp.sdn.sdn-pdt-api
218  com.hp.sdn.sdn-pdt
219  com.hp.sdn.sdn-pdt
220  com.hp.sdn.sdn-tvu
221  com.hp.sdn.sdn-tvu
222  com.hp.sdn.sdn-tvu
223  com.hp.sdn.sdn-ctl-
224  com.hp.sdn.sdn-app
232  com.hp.hm.hm-mod
233  com.hp.hm.hm-api
234  com.hp.hm.hm-dao-api
235  com.hp.hm.hm-dao-model
236  com.hp.hm.hm-dao
237  com.hp.hm.hm-bl
238  com.hp.hm.hm-rs
```

Service(s) between Bundles 237 and 238

Service [318] com.hp.hm.api.SwitchService
Published by Bundle 237
Used by Bundle 238
Also used by Bundles
66
ObjectClass
com.hp.hm.api.SwitchService

[160] com.hp.sdn.sdn-adm-bl
1.11.0.SNAPSHOT

[237] com.hp.hm.hm-bl
1.0.0.SNAPSHOT

[66] org.eclipse.virgo.shell.command
3.6.1.RELEASE

[238] com.hp.hm.hm-rs
1.0.0.SNAPSHOT

From bundle [237] com.hp.hm.hm-bl
1.0.0.SNAPSHOT
To bundle [238] com.hp.hm.hm-rs
1.0.0.SNAPSHOT

Virgo Admin Console from EclipseRT

© Copyright 2008, 2012 VMware Inc. Licensed under the Eclipse Public License v1.0.

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 [35] 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 dependency 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 115).

Dependent SwitchManager.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;
    }
```

```
        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 such 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:

```
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.ReferencePolicy;
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 gets available and *unbindAlertService(AalertService)* 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] [34] 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 Consolec on page 128. 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 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 [18], 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 6 describes the REST API implemented to expose the *SwitchService* functionality from the sample application.

Table 6 Switch REST API

Request	Description
GET /sdn/hm/v1.0/switches/	Lists all switches managed by the application.
GET /sdn/hm/v1.0/switches/{id}	Gets the switch with the given id.
POST /sdn/hm/v1.0/switches/	Adds a switch. The request's data must contain the switch data in JSON format.
PUT /sdn/hm/v1.0/switches/{id}	Updates a switch. The request's data must contain the switch

	data in JSON format.
DELETE /sdn/hm/v1.0/switches/{id}	Deletes the switch with the given identity.

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 115).

SwitchResource.java (REST API):

```
package com.hp.hm.rs;


import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.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 list() {
        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();
    }

    @PUT
    @Path("{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response update(@PathParam("id") long id, 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>
  <dependency>
    <groupId>com.sun.jersey.jersey-test-framework</groupId>
    <artifactId>jersey-test-framework-grizzly</artifactId>
    <version>1.17</version>
    <scope>test</scope>
  </dependency><dependency>
    <roupId>com.sun.jersey</groupId>
    <artifactId>jersey-servlet</artifactId>
    <version>1.17</version>
  </dependency>
```

The *hm-rs* module needs to be modified so it produces a web application archive (.war file) [36] 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 [2] that handles *HTTP* requests and dispatches to the right REST API based on the *@Path* annotations. The highlighted text in the next listing emphisizes the way the application's RESTful web services are registered within the Jersey Servlet.

REST Module Web Application (web.xml):

```
<?xml 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>
        <init-param>
            <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>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

Next, update the *hm-rs* module POM file *hm-rs/pom.xml* with the extract shown in the following listing to generate the .war file during the build process.

hm-rs/pom.xml to generate .war:

```
...
<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>
                <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,
                        com.hp.util.rs.auth,
                        com.hp.sdn.rs.misc,*
                    </Import-Package>
                    <Export-Package>!${banned.rs.paths}</Export-Package>
                    <Webapp-Context>${webapp.context}</Webapp-Context>
                    <Web-ContextPath>${web.context.path}</Web-ContextPath>
                </instructions>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-war-plugin</artifactId>
            <version>2.2</version>
            <configuration>
                <packagingExcludes>WEB-INF/lib/*.jar</packagingExcludes>
                <attachClasses>true</attachClasses>
                <webResources>
                    <resource>
                        <directory>target/scr-plugin-generated</directory>
                    </resource>
                </webResources>
                <archive>
                    <manifestFile>${project.build.directory}/META-
INF/MANIFEST.MF</manifestFile>
                    <manifestEntries>
                        <Bundle-ClassPath>WEB-INF/classes</Bundle-ClassPath>
                    </manifestEntries>
                </archive>
            </configuration>
        </plugin>
    </plugins>
</build>
</project>
```

Now update the application deployment plan *hm-app/hm.plan* (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 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-dao.api"
version="1.0.0.SNAPSHOT "/>
    <artifact type="bundle" name="com.hp.hm.hm-dao.model"
version="1.0.0.SNAPSHOT "/>
    <artifact type="bundle" name="com.hp.hm.hm-dao" 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
"/>
</plan>
```

Finally update the application packaging POM file *hm-app/pom.xml* (created in Application Packaging POM File on page 111) 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="${user.home}/.m2/repository/com/hp/hm/">
                                    <!— Add an <include> node for api, bl, dao-
api, dao-model and dao -->
                                    <include name="hm-
model/${project.version}/hm-model-${project.version}.jar"/>
```

```
                                                                <include name="hm-rs/${project.version}/hm-
rs-${project.version}.war"/>

                                    </fileset>
                                    <fileset dir="${basedir}" includes="hm.plan"/>
                                </copy>
                                <zip destfile="target/hm-${project.version}.zip"
basedir="target/bundles"/>
                            </tasks>
                        </configuration>
                        <goals>
                            <goal>run</goal>
                        </goals>
                    </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 137 using Curl [18]. See Curl on page 8 for installation instructions. Build and install the application as described in Building Application on page 115 and Installing Application on page 116.

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 6. Figure 52 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.

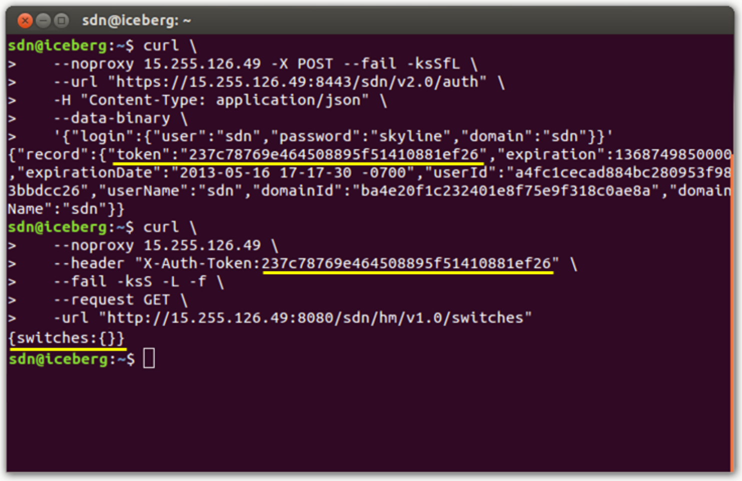
CURL Authentication Command:

```
$ curl --noproxy [SDN_CONTROLLER_ADDRESS] -X POST --fail -ksSfL \
   --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"
```

Figure 52 REST API CURL Execution Example



```
sdn@iceberg: ~
sdn@iceberg:~$ curl \
>     --noproxy 15.255.126.49 -X POST --fail -ksSfL \
>     --url "https://15.255.126.49:8443/sdn/v2.0/auth" \
>     -H "Content-Type: application/json" \
>     --data-binary \
>     '{"login":{"user":"sdn","password":"skyline","domain":"sdn"}}'
{"record":{"token":"237c78769e464508895f51410881ef26","expiration":1368749850000
,"expirationDate":"2013-05-16 17-17-30 -0700","userId":"a4fc1cecad884bc280953f98
3bbdcc26","userName":"sdn","domainId":"ba4e20f1c232401e8f75e9f318c0ae8a","domain
Name":"sdn"}}
sdn@iceberg:~$ curl \
>     --noproxy 15.255.126.49 \
>     --header "X-Auth-Token:237c78769e464508895f51410881ef26" \
>     --fail -ksS -L -f \
>     --request GET \
>     -url "http://15.255.126.49:8080/sdn/hm/v1.0/switches"
{switches:{}}
sdn@iceberg:~$ 
```

RESTful Web Services Unit Test

Even though at this point implementations use fake data, 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 115).

SwitchResourceTest.java:

```
package com.hp.hm.rs;

import com.hp.sdn.rs.misc.ControllerResourceTest;
import com.hp.util.rs.ResourceTest;
...
public 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.setDefaultMediaType(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 = "{\”switches\”:[]}";
    assertResponseContains(response, expectedResponse);
}

@Test
public void testGet() {
    long idMock = 1;
    String path = BASE_PATH + "/" + idMock;
    String response = get(path);
    String expectedResponse = "{\”switch\”:{}}";
    assertResponseContains(response, expectedResponse);
}

@Test
public void testAdd() {
    String jsonRequest = "{\”switch\”:{}}";
    String response = post(BASE_PATH, jsonRequest);
    String expectedResponse = "{switch:{}}";
    assertResponseContains(response, expectedResponse);
}

@Test
public void testUpdate() {
    long idMock = 1;
    String path = BASE_PATH + "/" + idMock;
    String jsonRequest = "{\”switch\”:{}}";
    String response = put(path, jsonRequest);
    String expectedResponse = "{\”switch\”:{}}";
    assertResponseContains(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>
<dependency>
  <groupId>commons-configuration</groupId>
  <artifactId>commons-configuration</artifactId>
  <version>1.6</version>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.1.4</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 [37] format in our sample application.
2. Call domain services—(Business Logic) to serve the request.
3. Encode the result to include in the response—JSON [37] 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 126). 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

137). 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 [35]. However, it is necessary to write an OSGi compliant service that subscribes/unsubscribes our Domain Services to/from the repository. Create the *ServiceAssistant* lass shown in the following listing under *hm-rs* module.

ServiceAssistant.java:

```
package com.hp.hm.rs;

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 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 134 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 137) that makes use of the inherited *get(Class<?>)* method to get a reference of *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.find(null, null);
        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 146) that uses *EasyMock* [38] 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.find(EasyMock.isNull(SwitchFilter.class),
            EasyMock.isNull(SortSpecification.class))).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 in previously the tasks a REST API normally accomplishes is decoding the request and encode the result into the response. This sample application uses JSON [37] 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. 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 *hm-rs* 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 119).

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;
}

@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 *hm-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 115).

JSON Module Dependencies:

```
<dependency>
  <groupId>com.hp.util</groupId>
  <artifactId>hp-util-codec</artifactId>
  <version>${hp-util.version}</version>
</dependency>
<dependency>
  <groupId>com.hp.sdn</groupId>
  <artifactId>sdn-adm-api</artifactId>
  <version>${sdn.version}</version>
</dependency>
```

The HP VAN SDN Controller SDK uses Jackson API [39] 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:

```
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 Factories (*JsonFactory*) services annotated with the following property: *"name=flare"*.

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:

```
    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:

```
    <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:

```
    package com.hp.hm.rs;
```

```
import com.hp.sdn.json.JsonService;
...
public class SwitchResourceTest extends ControllerResourceTest {

    private static final String BASE_PATH = "switches";

    private SwitchService switchServiceMock;
    private JsonService jsonServiceMock;

    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());

        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

        // Note that the expected switches can be anything
        // (it doesn't matter) since the SwitchService has been mocked.
        List<Switch> switches = Collections.emptyList();
        // Note that the returned JSON does not matter since
        // the JSON codec has been mocked.
```

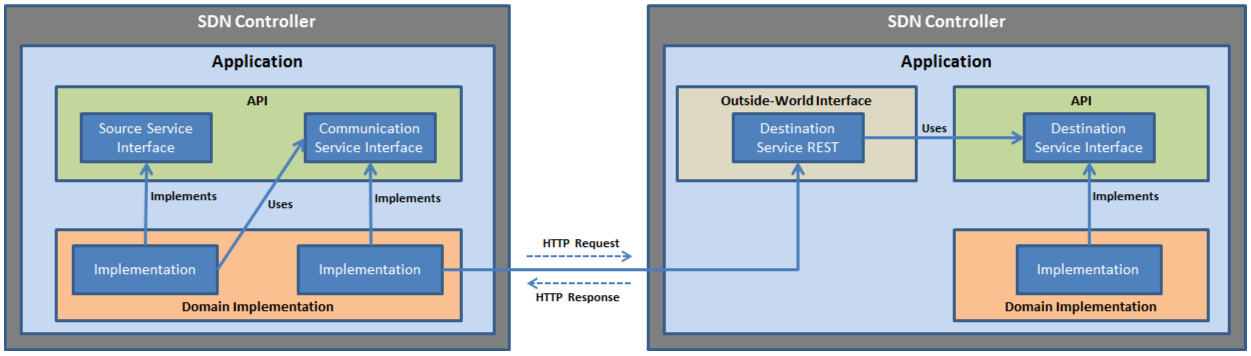
```
            String switchesJson = "{\"switches\":[]}";

            // 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) [2] [1] 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 ZooKeeper [10] to achieve that. For more information see HA Service on page 67.

Figure 53 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 such new communication service it is possible to define higher-level (type-safe) communication methods.

Figure 53 Controller-Controller Communication via REST (Sideway API)



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:

```
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:

```
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 response = restClient.get(restClient.login(), uri);

            String responseData;
            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(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 from *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 115).

ServiceRest Dependency:

```
<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 134. 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 126 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 than *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 145 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 [22].

One big advantage of the 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 [37] schema to express our 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:

```
{
    "com.hp.hm.model.Switch":
    { "properties":
        {
        "id": {"type": "long"},
        "mac_address": {"type": "string"},
        "ip_address": {"type": "string"},
        "friendly_name": {"type": "string"},
        "active_state": {"type": "string"}
        }
    }
}
```

Create a class to register the RESTI 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 115).

DocProvider.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:

```
<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 llisting. 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.

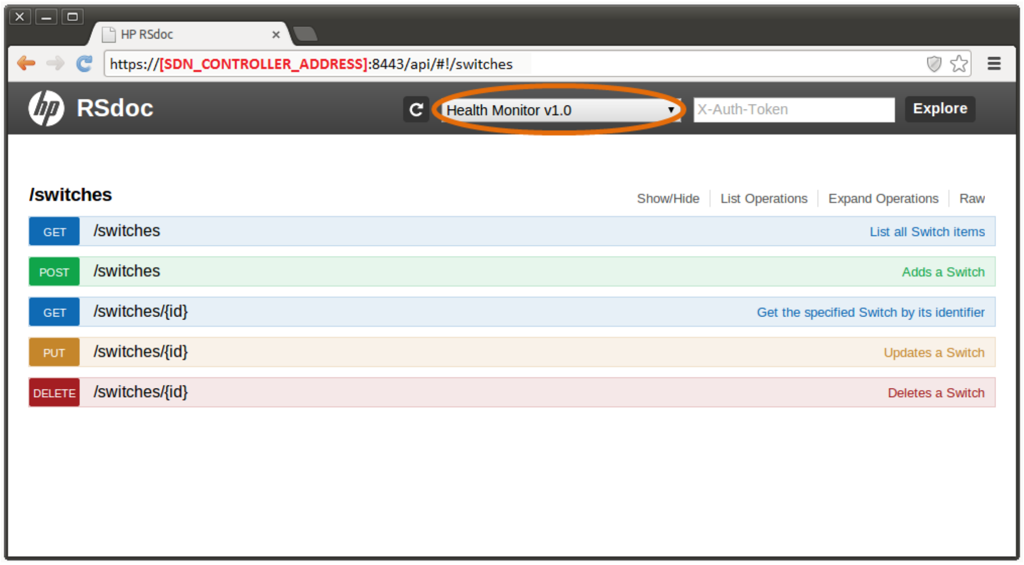
RSdoc Generation Maven Configuration:

```
    ...
    <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/${webapp.context}</api.url>
    </properties>
    ...
    <build>
        <plugins>
            <plugin>
                <artifactId>maven-antrun-plugin</artifactId>
                <executions>
                    <execution>
                        <id>generate-resources</id>
                        <phase>process-resources</phase>
                        <configuration>
                            <tasks>
                                <delete dir="target/classes/rsdoc" />
                                <mkdir dir="target/classes/rsdoc" />
                                <exec executable="java">
                                    <arg value="-Dapi.name=${api.name}" />
                                    <arg  value="-Dapi.version=${api.version}"
/>
                                    <arg value="-Dapi.url=${api.url}" />
                                    <arg value="-jar" />
                                    <arg
value="${user.home}/.m2/repository/com/hp/util/hp-util-rsdoc/${hp-
util.version}/hp-util-rsdoc-${hp-util.version}.jar" />
                                    <arg value="com/hp/hm/rs" />
                                    <arg value="target/classes/rsdoc" />
                                    <arg value="src/main/java" />
                                </exec>
                            </tasks>
                        </configuration>
```

```
                            <goals>
                                <goal>run</goal>
                            </goals>
                        </execution>
                    </executions>
                </plugin>
            ...
            </plugins>
        </build>
    </project>
```

Build and install the application as described Building Application on page 115 and Installing Application on page 116. RSdoc is now accessible as illustrated in Figure 54.

Figure 54 Sample Application RSdoc



/switches

Show/Hide | List Operations | Expand Operations | Raw

GET	/switches	List all Switch items
POST	/switches	Adds a Switch
GET	/switches/{id}	Get the specified Switch by its identifier
PUT	/switches/{id}	Updates a Switch
DELETE	/switches/{id}	Deletes a Switch

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 157) it may be annotated with *@RsDocIgnore* as illustrated in the following listing.

RsDocIgnore Annotation:

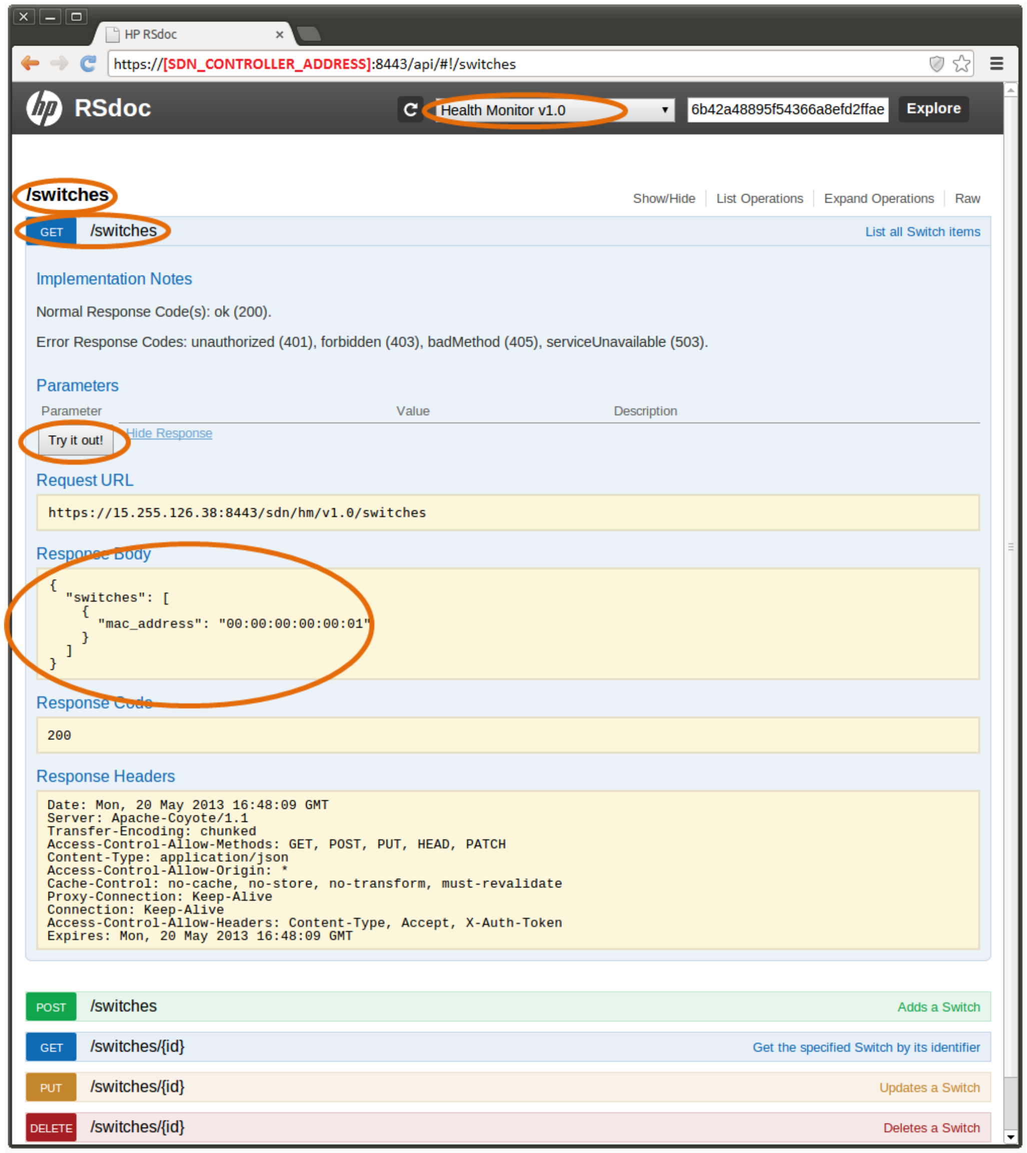
```
package com.hp.hm.rs;

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 16 to open Rsdoc and authenticate, and then try the sample application's REST API as illustrated in Figure 55. Modify *SwitchManager* to return some fake data in *find(SwitchFilte, SortSpecification<SwitchSortKey>)* method and try *GET* switches from the RSdoc.

Figure 55 Trying Application's REST API with RSdoc Example



Implementation Notes

Normal Response Code(s): ok (200).

Error Response Codes: unauthorized (401), forbidden (403), badMethod (405), serviceUnavailable (503).

Parameters

Request URL

```
https://15.255.126.38:8443/sdn/hm/v1.0/switches
```

Response Body

```
{
  "switches": [
    {
      "mac_address": "00:00:00:00:00:01"
    }
  ]
}
```

Response Code

```
200
```

Response Headers

```
Date: Mon, 20 May 2013 16:48:09 GMT
Server: Apache-Coyote/1.1
Transfer-Encoding: chunked
Access-Control-Allow-Methods: GET, POST, PUT, HEAD, PATCH
Content-Type: application/json
Access-Control-Allow-Origin: *
Cache-Control: no-cache, no-store, no-transform, must-revalidate
Proxy-Connection: Keep-Alive
Connection: Keep-Alive
Access-Control-Allow-Headers: Content-Type, Accept, X-Auth-Token
Expires: Mon, 20 May 2013 16:48:09 GMT
```

NOTE

The tool offered by the HP VAN SDN Controller SDK that generates the RSdoc takes the Javadoc [22] 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 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 52.

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-message-style {
    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 52 to find a SKI reference application that provides examples of SKI widgets). Create the file *hm-ui/src/main/webapp/js/hm-switches.js* with the content from the following listing, hm-switches.js. Create one JavaScript file for each view in the application.

hm-switches.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-switches.js');

    // Create a view with a toolbar button
    function load(view) {
        v.setToolbar(def.tbButton(view.mkId('btn'),
view.lion('key-button'), '', function () {
    v.setContent($('<span/>').
        addClass('hm-message-style').append('Hello World'));
        }));
    }

    // Adds the view to the framework with 'hm-switches' as its name.
    // The associated property file must have the same name than as view.
    def.addView('hm-switches', {
        load: load
    });

}(SKI));
```

Now create a script that adds a menu entry to the navigation panel so the *hm-switches* 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:

```
// 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 the "Health Monitor" (key-cat-hm) category to the
    // navigation panel and adds a "Switches" (key-nav-switches)
    // sub-menu pointing to hm-switches.js view.
    nav.insertCategoryAfter('c-tasks', 'key-cat-hm', [
        nav.item('key-nav-switches', 'hm-switches', 'square')
    ]);

}(SKI));
```

Except for "*Hello World*", plan 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 52 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-switches.js,* name it "*hm-switches*" (Highlighted in the hm-switches.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-switches.properties* with the content from the following hm-switches.properties listing. The prefix "*key-*" was added to the text keys to differentiate them from other strings used in the JavaScript code, however, follow the naming conventions defined in GUI on page 52. 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 52 for details about available icons and how define custom icons.

hm-switches.properties:

```
title = Open Flow Switches
icon = grid
key-button = Refresh Data
```

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
key-cat-hm = Health Monitor
key-nav-switches = Switches
```

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 llisting to the <dependencies> node; after updating the POM file update the Eclipse project dependencies (see Updating Project Dependencies on page 115).

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>
  <classifier>tests</classifier>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>com.hp.util</groupId>
  <artifactId>hp-util-skis</artifactId>
  <version>${hp-util.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.hml* 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:

```
<link href="/sdn/ui/hm/css/hm.css" rel="stylesheet">
```

UIExtension js.html :

```
<script src="/sdn/ui/hm/js/hm-nav.js"></script>
<script src="/sdn/ui/hm/js/hm-switches.js"></script>
```

The prefix "*sdn/ui/hm*" 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-switches.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) [36].

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 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 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:

```
...
<modelVersion>4.0.0</modelVersion>
<artifactId>hm-ui</artifactId>
<packaging>war</packaging>
...
<properties>
    <banned.rs.paths>com.hp.hm.ui</banned.rs.paths>
    <webapp.context>sdn/ui/hm</webapp.context>
    <web.context.path>sdn/ui/hm<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>
            <configuration>
                <manifestLocation>${project.build.directory}/META-
INF</manifestLocation>
                <supportedProjectTypes>
                    <supportedProjectType>bundle</supportedProjectType>
                    <supportedProjectType>war</supportedProjectType>
                </supportedProjectTypes>
                <instructions>
                    <Import-Package>
                        com.sun.jersey.api.core,
                        com.sun.jersey.spi.container.servlet,
                        com.sun.jersey.server.impl.container.servlet,
                        com.hp.util.rs,
                        com.hp.util.rs.auth,
                        com.hp.sdn.rs.misc,
                        com.hp.sdn.ui.misc,*
                    </Import-Package>
                    <Export-Package>!${banned.rs.paths}</Export-Package>
                    <Webapp-Context>${webapp.context}</Webapp-Context>
                    <Web-ContextPath>${web.context.path}</Web-ContextPath>
                </instructions>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-war-plugin</artifactId>
            <version>2.2</version>
            <configuration>
                <packagingExcludes>WEB-INF/lib/*.jar</packagingExcludes>
                <attachClasses>true</attachClasses>
                <webResources>
                    <resource>
                        <directory>target/scr-plugin-generated</directory>
                    </resource>
                </webResources>
                <archive>
                    <manifestFile>${project.build.directory}/META-
INF/MANIFEST.MF</manifestFile>
                    <manifestEntries>
                        <Bundle-ClassPath>WEB-INF/classes</Bundle-ClassPath>
                    </manifestEntries>
                </archive>
            </configuration>
        </plugin>
    </plugins>
    </build>
</project>
```

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:

```
<?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-dao.api"
version="1.0.0.SNAPSHOT "/>
    <artifact type="bundle" name="com.hp.hm.hm-dao.model"
version="1.0.0.SNAPSHOT "/>
    <artifact type="bundle" name="com.hp.hm.hm-dao" 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
"/>
    <artifact type="bundle" name="com.hp.hm.hm-ui" version="1.0.0.SNAPSHOT
"/>
</plan>
```

Finally update the application packaging POM file *hm-app/pom.xml* with the extract shown in the following listing to include the .war file into the application package.

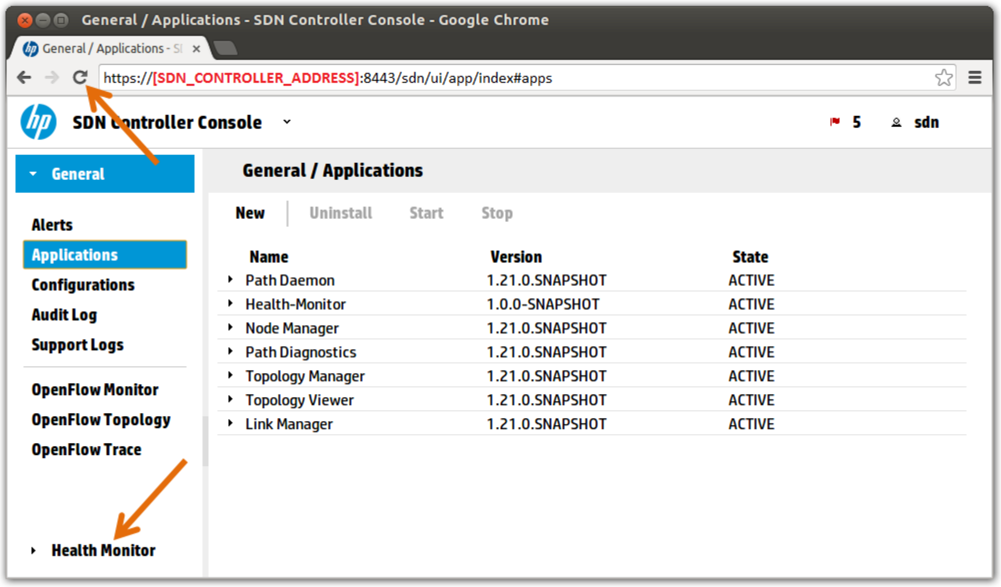
Sample Application Packaging POM File Including UI Module:

```
...
<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"/>
```

```
                                    <!— Add an <include> node for api, bl,
dao-api, dao-model and dao -->
                                    <include name="hm-
rs/${project.version}/hm-rs-${project.version}.war"/>
                                    <include name="hm-
ui/${project.version}/hm-ui-${project.version}.war"/>
                                </fileset>
                                <fileset dir="${basedir}"
includes="hm.plan"/>
                            </copy>
                            <zip destfile="target/hm-${project.version}.zip"
basedir="target/bundles"/>
                        </tasks>
                    </configuration>
                    <goals>
                        <goal>run</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
...
```

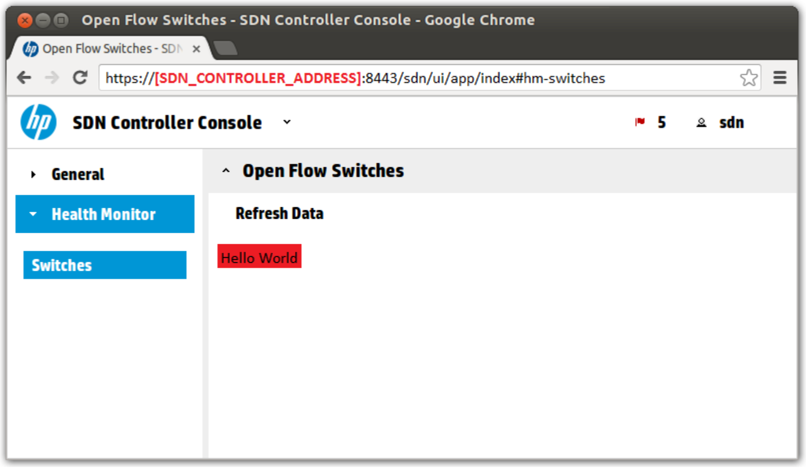
Now try the application's user interface. Build and install the application as described in Building Application on page 115 and Installing Application on page 116. After installing the application refresh the SDN Controller GUI as illustrated at the top part of Figure 56; the application's GUI entry appears as illustrated at the bottom part of Figure 56. Figure 57 shows the application's view after clicking the "*Refresh Data*" button.

Figure 56 Sample Application GUI Entry



Name	Version	State
Path Daemon	1.21.0.SNAPSHOT	ACTIVE
Health-Monitor	1.0.0-SNAPSHOT	ACTIVE
Node Manager	1.21.0.SNAPSHOT	ACTIVE
Path Diagnostics	1.21.0.SNAPSHOT	ACTIVE
Topology Manager	1.21.0.SNAPSHOT	ACTIVE
Topology Viewer	1.21.0.SNAPSHOT	ACTIVE
Link Manager	1.21.0.SNAPSHOT	ACTIVE

Figure 57 Sample Application View


Open Flow Switches - SDN Controller Console - Google Chrome
Open Flow Switches - SDN
https://[SDN_CONTROLLER_ADDRESS]:8443/sdn/ui/app/index#hm-switches
SDN Controller Console
5
sdn
General
Health Monitor
Switches
Open Flow Switches
Refresh Data
Hello World

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 137 under the "REST Module Dependencies" listing and the RESTful Web Services Unit Test on page 146 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 115).

The following SwitchViewResource.java listing shows how to create the Switch View REST API named *SwitchViewResource*. The SwitchViewResource.java listing shows an extract of the resource. To use JSON encoding see JSON Encoding on page 152. To write unit test follow the instructions from RESTful Web Services Unit Test on page 146.

SwitchViewResource.java:

```
package com.hp.hm.ui.rs;
...
@Path("switches")
public class SwitchViewResource extends ControllerResource {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public Response hello() {
        // SwitchService is used just to illustrate that
        // services are available.
        SwitchService service = get(SwitchService.class);
        List<Switch> switches = service.find(null, null);
        return ok("Hello from the HP SDN Controller: " +
            switches.toString()).build();
    }
}
```

Now update the web.xml placeholder added in Module Configuration on page 168 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 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 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.SwitchViewResource

                <!— 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>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 165 under the “hm-switches.js” listing to make the remote call as illustrated in the following listing:

hm-switches.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('key-button'),
'', function () {
            $.get('/sdn/ui/hm/app/rs/switches', function(data) {
                v.setContent($('<span/>').addClass("hm-message-
style").append(data));
            });
        }));
    }
    ...
```

As seen in the code the view connects to the relative path *"/sdn/ui/hm/app/rs/switches"* so the connection is opened to the same controller that generated the web page. The prefix *"/sdn/ui/hm"* must match the *web.context.path* property defined in Module Configuration on page 168 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 173 under the "UI Module Web Application (web.xml) Defining Jersey Servlet" listing, and the suffix "*switches*" is the relative path of the resource given by the *@Path* annotation in the GUI-Specific REST API on page 173 under the "SwitchViewResource.java" listing.

In this case, the response media type (in *SwitchViewResource*) is defined as TEXT_PLAN. This means that the data parameter of the $.get() function callback is filled in with a plan 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 when 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 Application on page 115 and Installing Application on page 116. After installing the application refresh the SDN Controller GUI as illustrated at the top part of Figure 56; the application's GUI entry will appear as illustrated at the bottom part of Figure 56. Now the message returned by *SwitchViewResource* 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 declareative services as depicted in section Consuming Services with OSGi Declarative Services on page 134.

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 148, 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 busness 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-thread 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:

```
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, Long>, Switch> devices;
    private AtomicLong idCount;

    public SwitchManager(SystemInformationService systemInformationService) {
        if (systemInformationService == null) {
            throw new NullPointerException(...);
        }
        this.systemInformationService = systemInformationService;

        devices = new HashMap<Id<Switch, Long>, Switch>();
        idCount = new AtomicLong(1);
    }

    public void setAlertService(AlertService alertService) {
        this.alertService = alertService;
    }

    @Override
    public Switch add(Switch device) {
        if (device == null) {
            throw new NullPointerException("device cannot be null");
        }

        Switch deviceToAdd = device;
        if (device.getId() == null) {
            Id<Switch, Long> id = Id.valueOf(idCount.getAndIncrement());
            deviceToAdd = new Switch(id, device.getMacAddress());
            deviceToAdd.setActiveState(device.getActiveState());
            deviceToAdd.setFriendlyName(device.getFriendlyName());
            deviceToAdd.setIpAddress(device.getIpAddress());
        }

        devices.put(deviceToAdd.getId(), deviceToAdd);

        return deviceToAdd;
    }

    @Override
    public void update(Switch device) {
        if (device == null) {
            throw new NullPointerException("device cannot be null");
        }

        if (device.getId() == null) {
            throw new IllegalArgumentException(
                "a device must be added before updating it");
        }

        devices.put(device.getId(), device);
    }

    @Override
    public Switch get(Id<Switch, Long> id) {
        if (id == null) {
            throw new NullPointerException("id cannot be null");
        }
        return devices.get(id);
    }

    @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
        return new ArrayList<Switch>(devices.values());
    }

    @Override
    public void delete(Id<Switch, Long> id) {
            if (id == null) {
                throw new NullPointerException("id cannot be null");
            }
            devices.remove(id);
        }
    }
```

SwitchResource.java Delegating to Business Logic:

```
    package com.hp.hm.rs;
    ...
    @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 json = jsonService.toJsonList(switches, Switch.class, true);

            return ok(json).build();
        }

        @GET
        @Path("{id}")
        @Produces(MediaType.APPLICATION_JSON)
        public Response get(@PathParam("id") long id) {
            Id<Switch, Long> deviceId = Id.valueOf(id);

            SwitchService service = get(SwitchService.class);
            Switch device = service.get(deviceId);

            JsonService jsonService = get(JsonService.class);
            String json = jsonService.toJson(device, true);

            return ok(json).build();
        }

        @POST
        @Produces(MediaType.APPLICATION_JSON)
        public Response add(String request) {
            JsonService jsonService = get(JsonService.class);
            Switch device = jsonService.fromJson(request, Switch.class);
            SwitchService service = get(SwitchService.class);
            device = service.add(device);
    String json = jsonService.toJson(device, true);
    return ok(json).build();
}

@PUT
@Path("{id}")
@Produces(MediaType.APPLICATION_JSON)
public Response update(@PathParam("id") long id, String request) {
    JsonService jsonService = get(JsonService.class);
    Switch source = jsonService.fromJson(request, Switch.class);

    SwitchService service = get(SwitchService.class);
    Id<Switch, Long> deviceId = Id.valueOf(id);
    Switch target = service.get(deviceId);

    if (target == null) {
        throw new NotFoundException(
            "device with id '" + id + "' not found");
    }

    target.setIpAddress(source.getIpAddress());
    target.setFriendlyName(source.getFriendlyName());
    target.setActiveState(source.getActiveState());

    service.update(target);
    String json = jsonService.toJson(target, true);

    return ok(json).build();
}

@DELETE
@Path("{id}")
@Produces(MediaType.APPLICATION_JSON)
public Response delete(@PathParam("id") long id) {
    Id<Switch, Long> deviceId = Id.valueOf(id);

    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();
```

```
        }
    }
```

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 updated with an unreachable status. See Alert Logging on page 19 to get more information.

At this point *SwitchManager* already depends on the *AlertService*, so it is ready to use such service. The following listing illustrates an extract of a modified *SwitchManager* that posts an alert when a device is unreachable.

SwitchManager.java Posting Alerts:

```
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 void update(Switch device) {
        if (device == null) {
            throw new NullPointerException("device cannot be null");
        }

        if (device.getId() == null) {
            throw new IllegalArgumentException(
                "a device must be added before updating it");
        }

        devices.put(device.getId(), device);

        if (alertService != null) {
            if (device.getActiveState() == ActiveState.OFF) {
                String source = "OpenFlow Switch: " +
                    device.getId().getValue();
                String data = "Active State: " + device.getActiveState();
```

```
                    alertService.post(Severity.WARNING, alertTopic,
                        source, data);
                }
            }
        }
        ...
    }
```

When the optional *AlertService* is set an alert topic is registered using the *AlertService*. Such 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

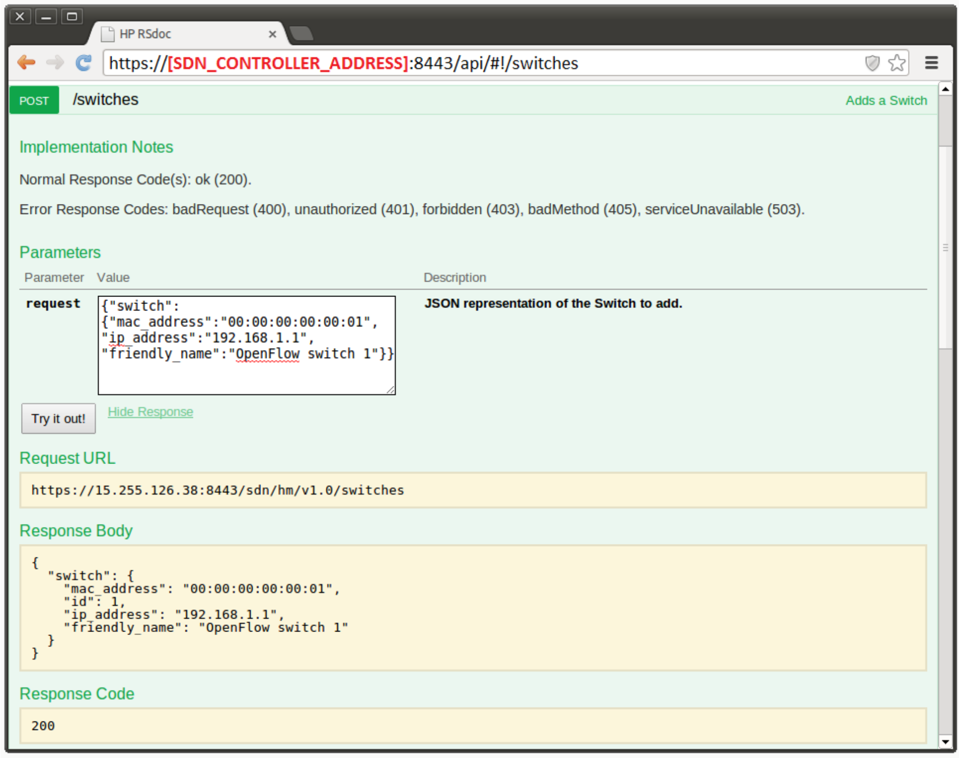
As mentioned before, a better design would make use of the decorator patter [XXX] to decorate business logic with optional dependencies so no check for null is needed and logics with different concerns are separated.

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 Application on page 115 and Installing Application on page 116.
2. Open the HP VAN SDN Controller's Rsdoc and authenticate as illustrated in Trying the REST API with RSdoc on page 164.
3. Add (POST) a device using the following JSON document: *{"switch":{"mac_address":"00:00:00:00:00:01","ip_address":"192.168.1.1","friendly_name":" OpenFlow switch 1"}}* (as illustrated in Figure 58)

Figure 58 Adding OpenFlow Switch



Implementation Notes

Normal Response Code(s): ok (200).

Error Response Codes: badRequest (400), unauthorized (401), forbidden (403), badMethod (405), serviceUnavailable (503).

Parameters

Parameter	Value	Description
request	{"switch":{"mac_address":"00:00:00:00:00:01", "ip_address":"192.168.1.1", "friendly_name":"OpenFlow switch 1"}}	JSON representation of the Switch to add.

Request URL

```
https://15.255.126.38:8443/sdn/hm/v1.0/switches
```

Response Body

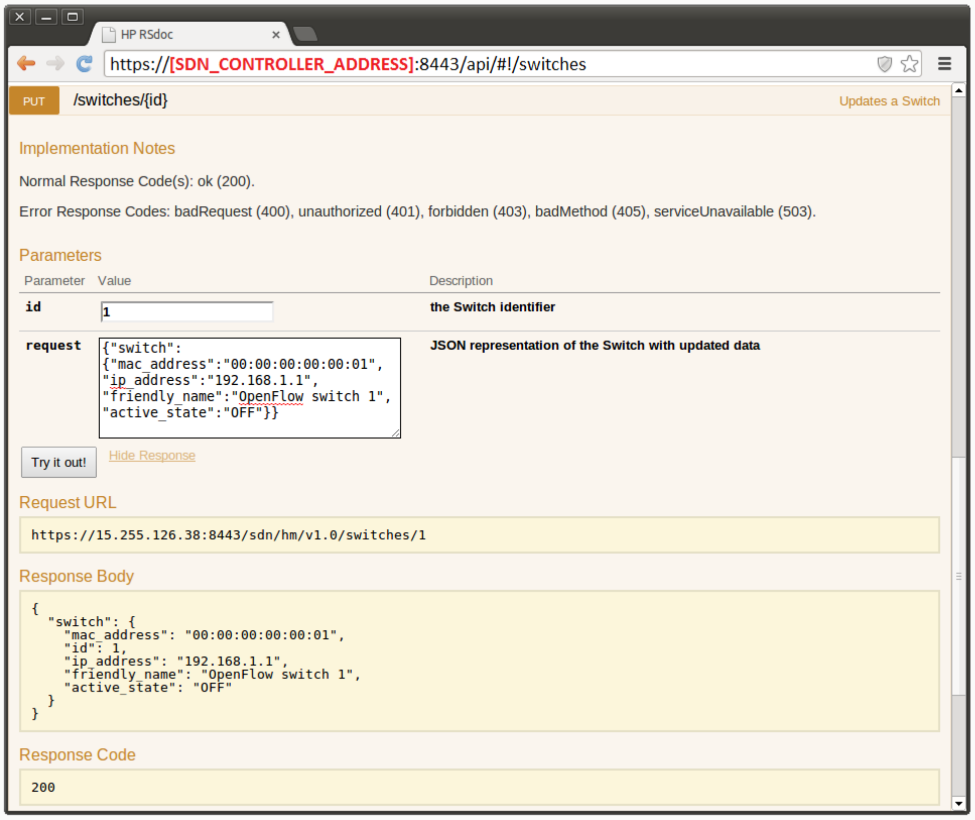
```
{
  "switch": {
    "mac_address": "00:00:00:00:00:01",
    "id": 1,
    "ip_address": "192.168.1.1",
    "friendly_name": "OpenFlow switch 1"
  }
}
```

Response Code

```
200
```

4. Update (PUT) the device using the following JSON document: *{"switch":{"mac_address":"00:00:00:00:00:01","ip_address":"192.168.1.1","friendly_name":"OpenFlow switch 1", "active_state":"OFF"}}* (as illustrated in Figure 59)

Figure 59 Updating OpenFlow Switch



Implementation Notes

Normal Response Code(s): ok (200).

Error Response Codes: badRequest (400), unauthorized (401), forbidden (403), badMethod (405), serviceUnavailable (503).

Parameters

Parameter	Value	Description
id	1	**the Switch identifier**
request	{"switch":{"mac_address":"00:00:00:00:00:01","ip_address":"192.168.1.1","friendly_name":"OpenFlow switch 1","active_state":"OFF"}}	**JSON representation of the Switch with updated data**

Request URL

```
https://15.255.126.38:8443/sdn/hm/v1.0/switches/1
```

Response Body

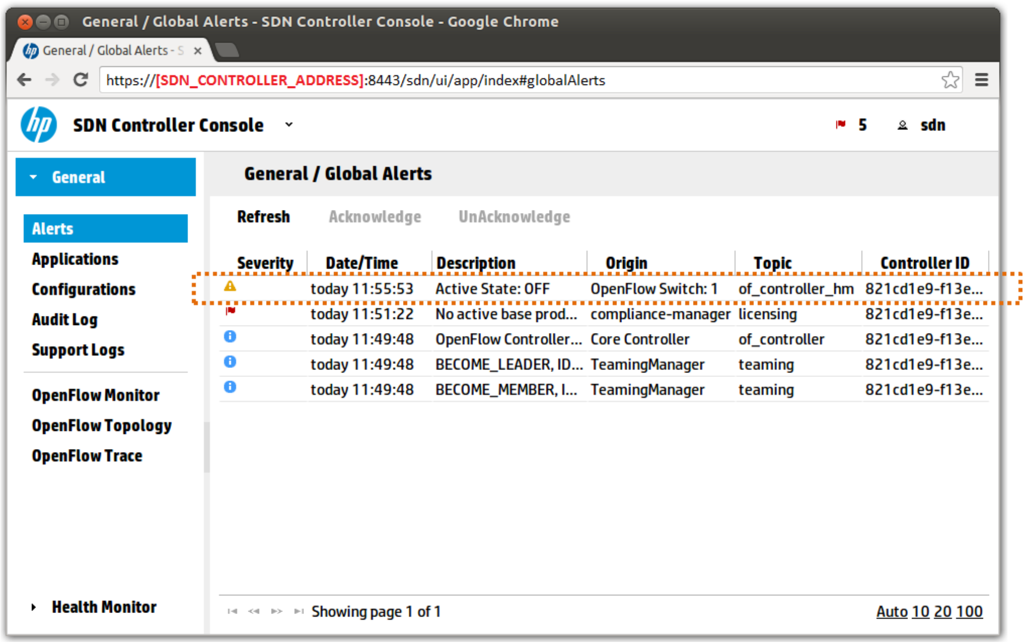
```
{
  "switch": {
    "mac_address": "00:00:00:00:00:01",
    "id": 1,
    "ip_address": "192.168.1.1",
    "friendly_name": "OpenFlow switch 1",
    "active_state": "OFF"
  }
}
```

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 60)

Figure 60 Alerts View



Severity	Date/Time	Description	Origin	Topic	Controller ID
⚠	today 11:55:53	Active State: OFF	OpenFlow Switch: 1	of_controller_hm	821cd1e9-f13e...
⚑	today 11:51:22	No active base prod...	compliance-manager	licensing	821cd1e9-f13e...
ℹ	today 11:49:48	OpenFlow Controller...	Core Controller	of_controller	821cd1e9-f13e...
ℹ	today 11:49:48	BECOME_LEADER, ID...	TeamingManager	teaming	821cd1e9-f13e...
ℹ	today 11:49:48	BECOME_MEMBER, I...	TeamingManager	teaming	821cd1e9-f13e...

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 18 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 134. The following listings illustrates an extract of a modified *SwitchManager* that posts audit logs.

SwitchManager.java Posting Audit Logs:

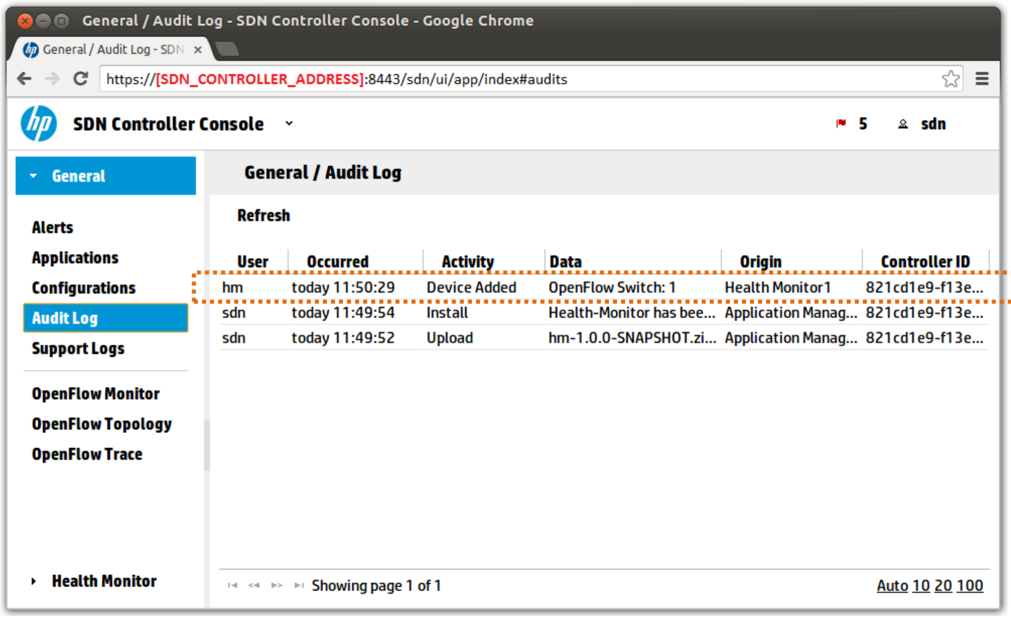
```
package com.hp.hm.impl;

import com.hp.sdn.adm.auditlog.AuditLogService;
...
public class SwitchManager implements SwitchService {
    ...
    private AuditLogService auditLogService;
    ...
    public void setAuditLogService(AuditLogService auditLogService) {
        this.auditLogService = auditLogService;
    }

    @Override
    public Switch add(Switch device) {
        ...
        devices.put(deviceToAdd.getId(), deviceToAdd);

        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 181 to add and modify an OpenFlow switch so an audit log is generated.

Figure 61 Audit Logs View



User	Occurred	Activity	Data	Origin	Controller ID
hm	today 11:50:29	Device Added	OpenFlow Switch: 1	Health Monitor1	821cd1e9-f13e...
sdn	today 11:49:54	Install	Health-Monitor has bee...	Application Manag...	821cd1e9-f13e...
sdn	today 11:49:52	Upload	hm-1.0.0-SNAPSHOT.zi...	Application Manag...	821cd1e9-f13e...

Debugging with Logs

The HP VAN SDN Controller uses the Simple Logging Facade 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 Usuing Logging:

```
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 [30] 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 Open Flow 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.

NOTE:

This may not be the best way to monitor reachabilioty status and such monitoring may not concern to SDN applications but 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 OpenFlow on page 22 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);
        } else {
            device.setActiveState(ActiveState.OFF);
        }
        update(device);
    }
}

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 bee seen that the MAC Address is used in to relate connected devices to the devices managed by the sample application.

Some dependencies need to resolved to use the OpenFlow controller services. 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 115).

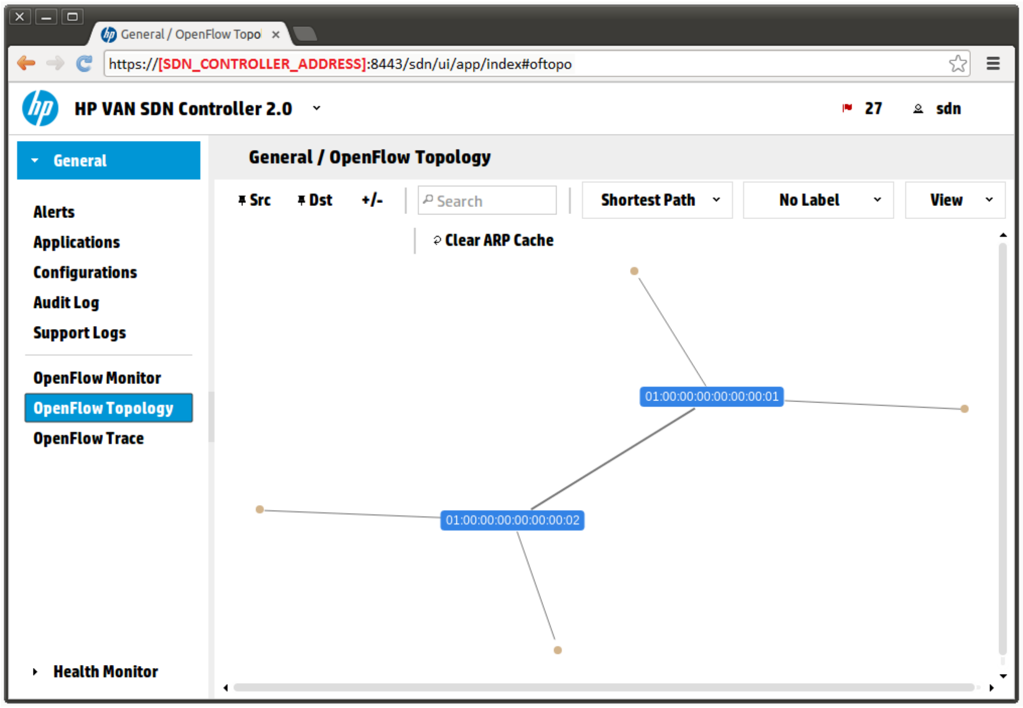
OpenFlow Controller Dependencies:

```
<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.

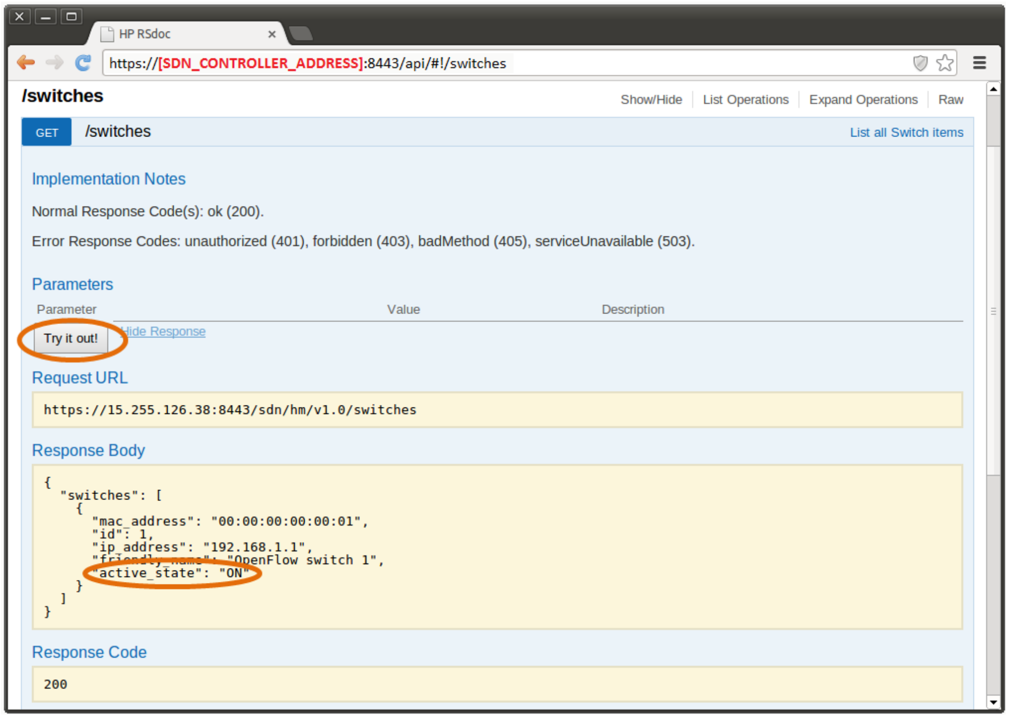
1. Follow the steps from Posting Alerts on page 181 to add an OpenFlow switch using the HP VAN SDN Controller's Rsdoc. Just add the device, no need to modify it since the active state will be automatically updated based on the device connectivity status. Make sure the MAC Address used in the posted JSON document (illustrated in Figure 58) 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 62. In this example two OpwnFlow-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 62 OpenFlow Topoligy View



3. Verify the active state of the device added in Step 1 has been updated using the Rsdoc as illustrated in Figure 63.

Figure 63 Sample Application OpenFlow Devices



Implementation Notes

Normal Response Code(s): ok (200).

Error Response Codes: unauthorized (401), forbidden (403), badMethod (405), serviceUnavailable (503).

Parameters

Parameter	Value	Description

Request URL

```
https://15.255.126.38:8443/sdn/hm/v1.0/switches
```

Response Body

```
{
  "switches": [
    {
      "mac_address": "00:00:00:00:00:01",
      "id": 1,
      "ip_address": "192.168.1.1",
      "friendly_name": "OpenFlow switch 1",
      "active_state": "ON"
    }
  ]
}
```

Response Code

```
200
```

After disconnecting the devices from the HP VAN SDN Controller (Stoping Mininet [45] in case of a virtualized network) the device's active state should be updated to OFF.

5 Testing Applications

The following information describes how to test SDN applications by executing Unit Test and enabling remote debugging in the controller.

Unit Testing

Unit test is automatically run when building the application; see Building Application on page 115. There is a version of this command to avoid running unit tests:

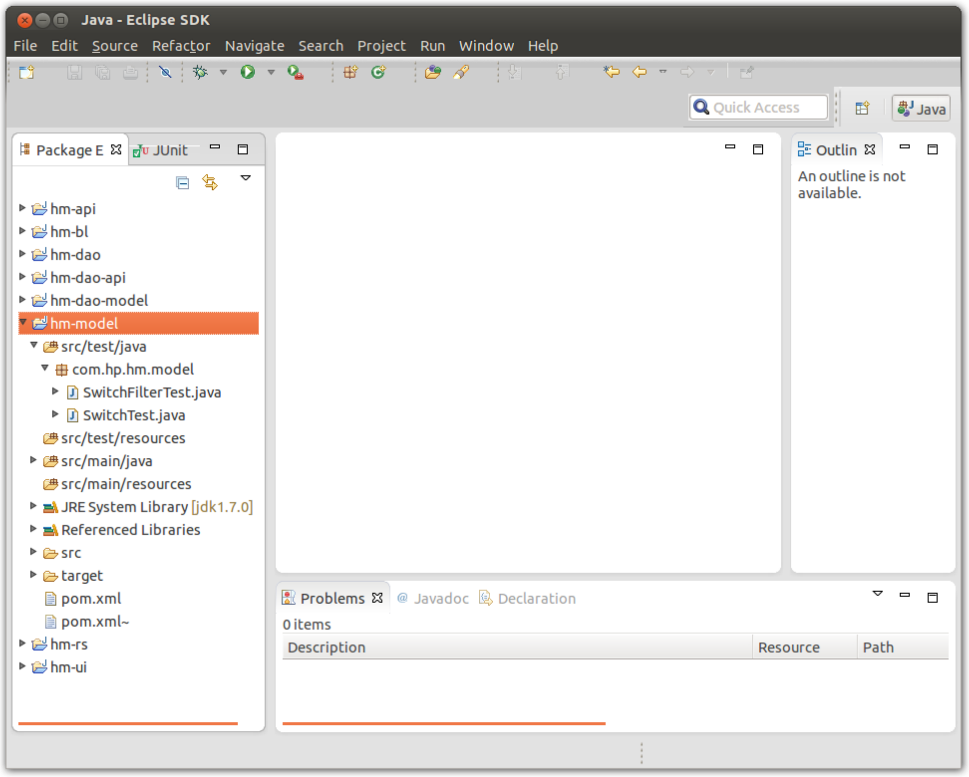
Building Application Ignoring Unit Test:

```
$ 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 64 and Figure 65.

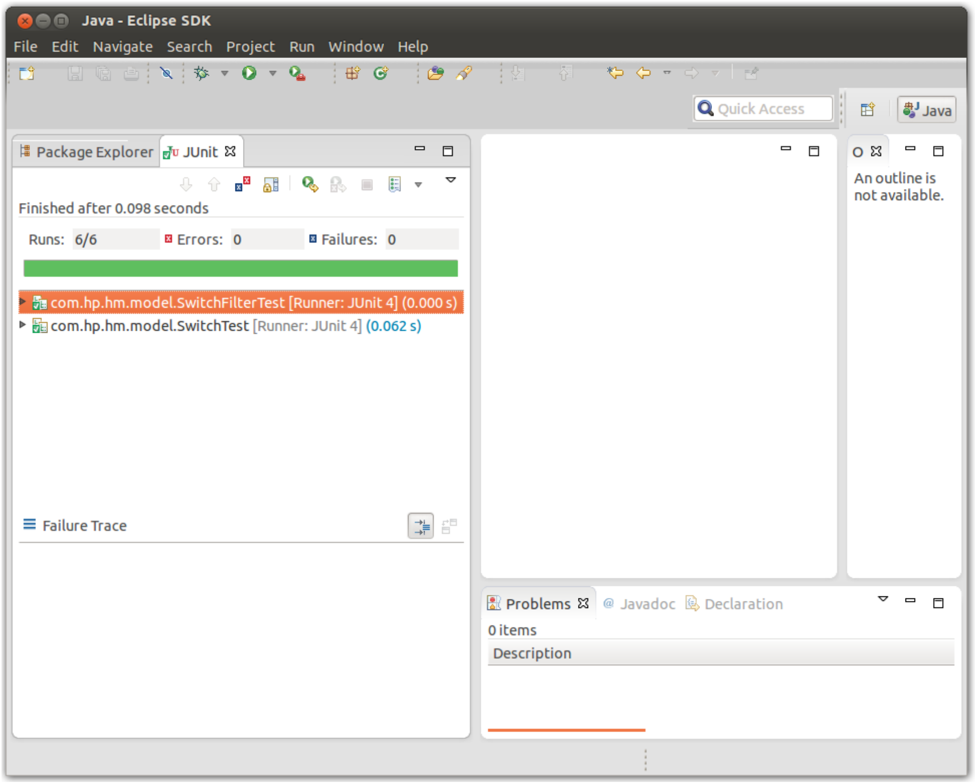
Figure 64 Running Unit Test within Eclipse (Step 1)



An outline is not available.

0 items

Figure 65 Running Unit Test within Eclipse (Step 2)



Finished after 0.098 seconds

Runs: 6/6 Errors: 0 Failures: 0

com.hp.hm.model.SwitchFilterTest [Runner: JUnit 4] (0.000 s)
com.hp.hm.model.SwitchTest [Runner: JUnit 4] (0.062 s)

An outline is not available.

0 items
Description

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 66 and Figure 67. See Installing Eclipse Plug-ins on page 211 to follow instructions about installing a plug-in; use *http://update.eclemma.org* as the repository location.

Figure 66 Unit Test Coverage

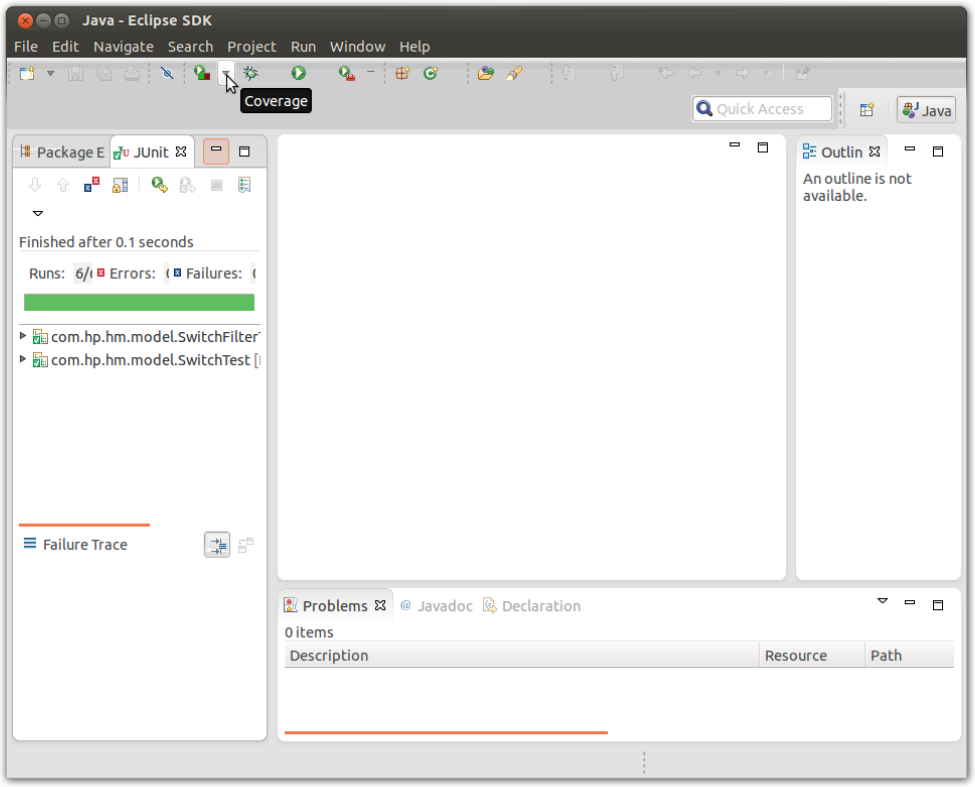
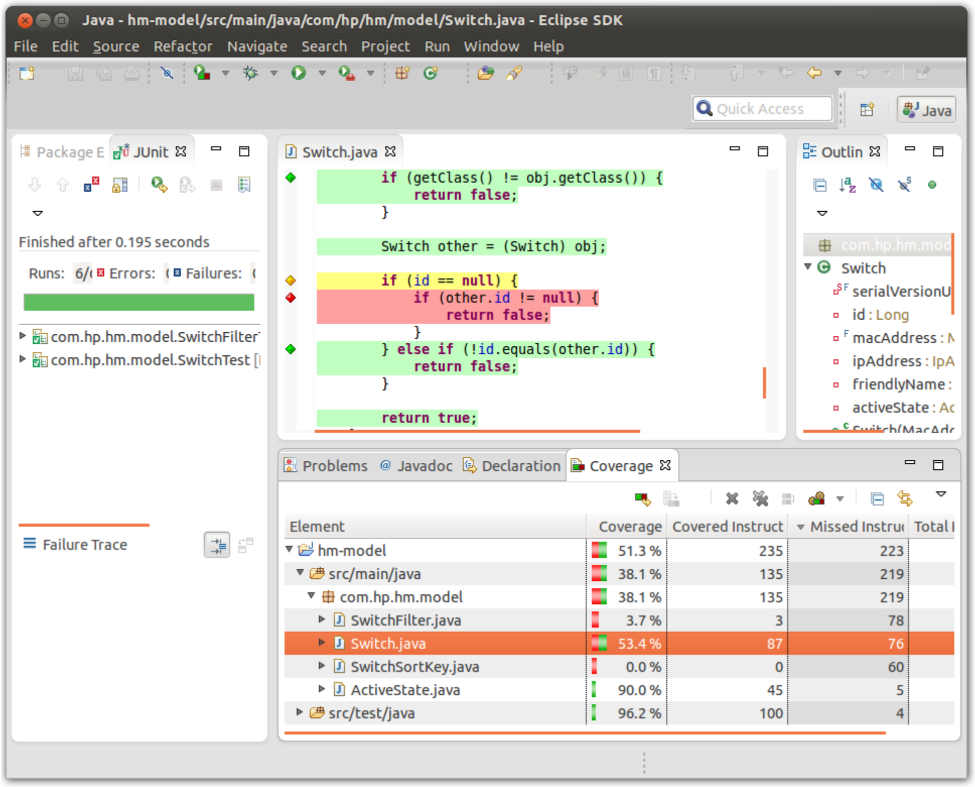

Java - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Coverage
Quick Access
Java
Package E
JUnit
Finished after 0.1 seconds
Runs: 6/6 Errors: 0 Failures: 0
com.hp.hm.model.SwitchFilter
com.hp.hm.model.SwitchTest
Failure Trace
Outlin
An outline is not available.
Problems
Javadoc
Declaration
0 items
Description
Resource
Path

Figure 67 Unit Test Coverage Result


Java - hm-model/src/main/java/com/hp/hm/model/Switch.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Java
Package E
JUnit
Finished after 0.195 seconds
Runs: 6/6 Errors: 0 Failures: 0
com.hp.hm.model.SwitchFilter
com.hp.hm.model.SwitchTest
Failure Trace
Switch.java
Outlin
com.hp.hm.mod
Switch
serialVersionU
id : Long
macAddress : M
ipAddress : IpA
friendlyName :
activeState : Ac
Problems
Javadoc
Declaration
Coverage

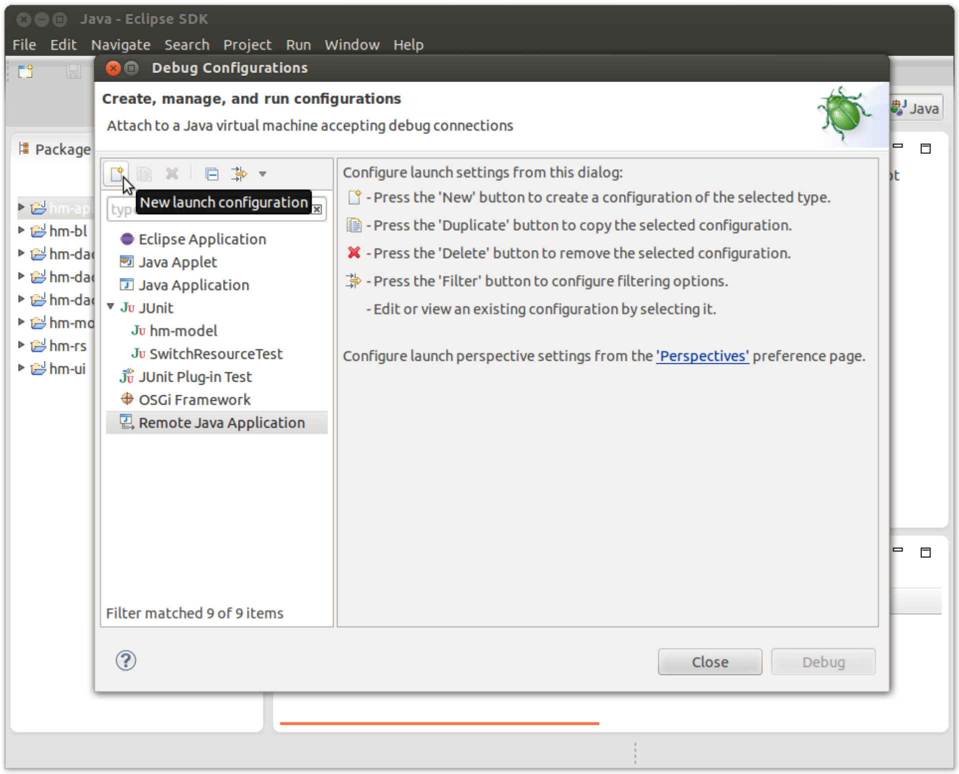
```
        if (getClass() != obj.getClass()) {
            return false;
        }

        Switch other = (Switch) obj;

        if (id == null) {
            if (other.id != null) {
                return false;
            }
        } else if (!id.equals(other.id)) {
            return false;
        }

        return true;
```

Element	Coverage	Covered Instruct	Missed Instruc	Total I
hm-model	51.3 %	235	223	
src/main/java	38.1 %	135	219	
com.hp.hm.model	38.1 %	135	219	
SwitchFilter.java	3.7 %	3	78	
Switch.java	53.4 %	87	76	
SwitchSortKey.java	0.0 %	0	60	
ActiveState.java	90.0 %	45	5	
src/test/java	96.2 %	100	4	

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 68.

Figure 68 Remote Java Application's Debug Configuration



Debug Configurations

Create, manage, and run configurations

Attach to a Java virtual machine accepting debug connections

- Eclipse Application
- Java Applet
- Java Application
- JUnit
  - hm-model
  - SwitchResourceTest
- JUnit Plug-in Test
- OSGi Framework
- Remote Java Application

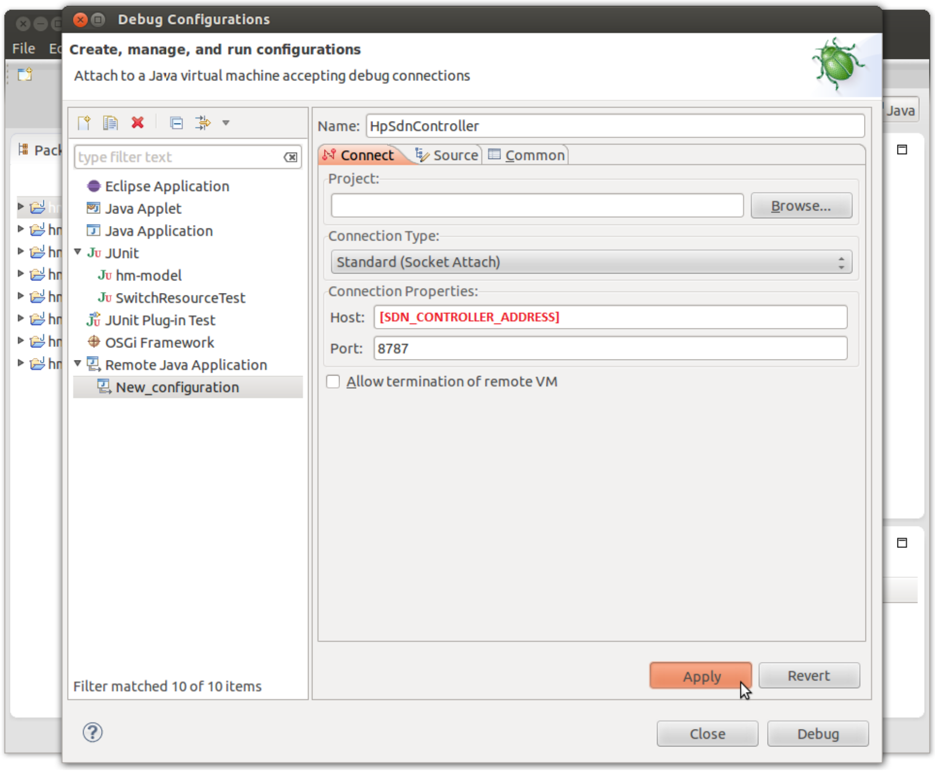
Filter matched 9 of 9 items

Configure launch settings from this dialog:
- Press the 'New' button to create a configuration of the selected type.
- Press the 'Duplicate' button to copy the selected configuration.
- Press the 'Delete' button to remove the selected configuration.
- Press the 'Filter' button to configure filtering options.
- Edit or view an existing configuration by selecting it.

Configure launch perspective settings from the 'Perspectives' preference page.

Set the SDN Controller configuration with the data shown in Figure 69. 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 70. 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 69 HP VAN SDN Controller's Remote Debug Configuration



Name: HpSdnController

Project:

Connection Type: Standard (Socket Attach)

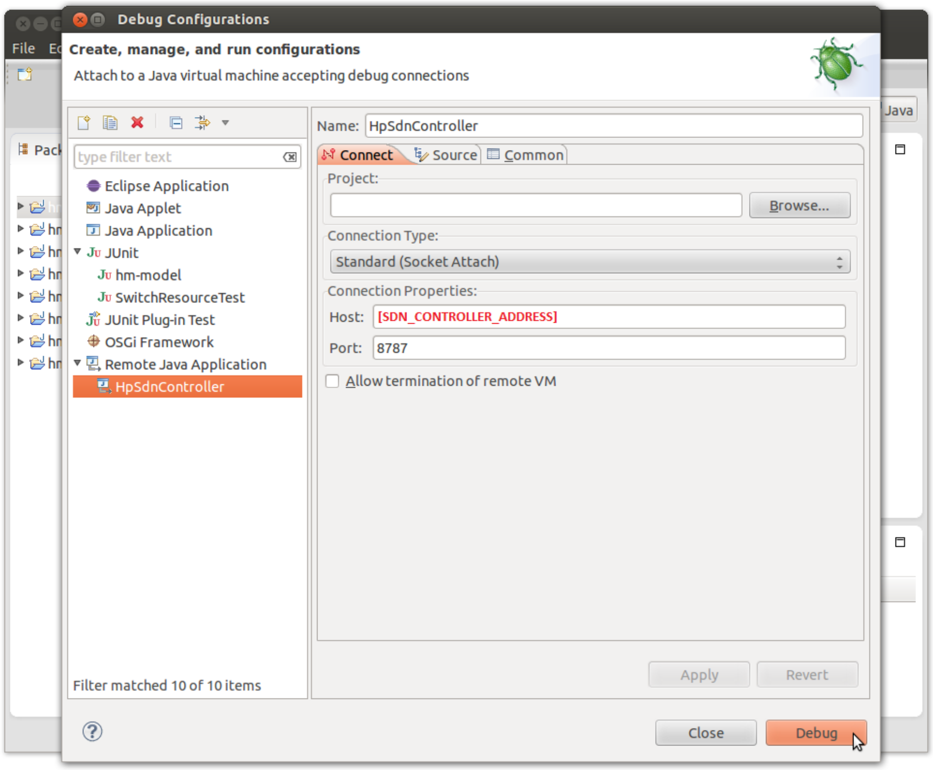
Connection Properties:

Host: [SDN_CONTROLLER_ADDRESS]

Port: 8787

☐ Allow termination of remote VM

Figure 70 HP VAN SDN Controller Saved Remote Debug Configuration



Name: HpSdnController

Project:

Connection Type: Standard (Socket Attach)

Connection Properties:

Host: [SDN_CONTROLLER_ADDRESS]

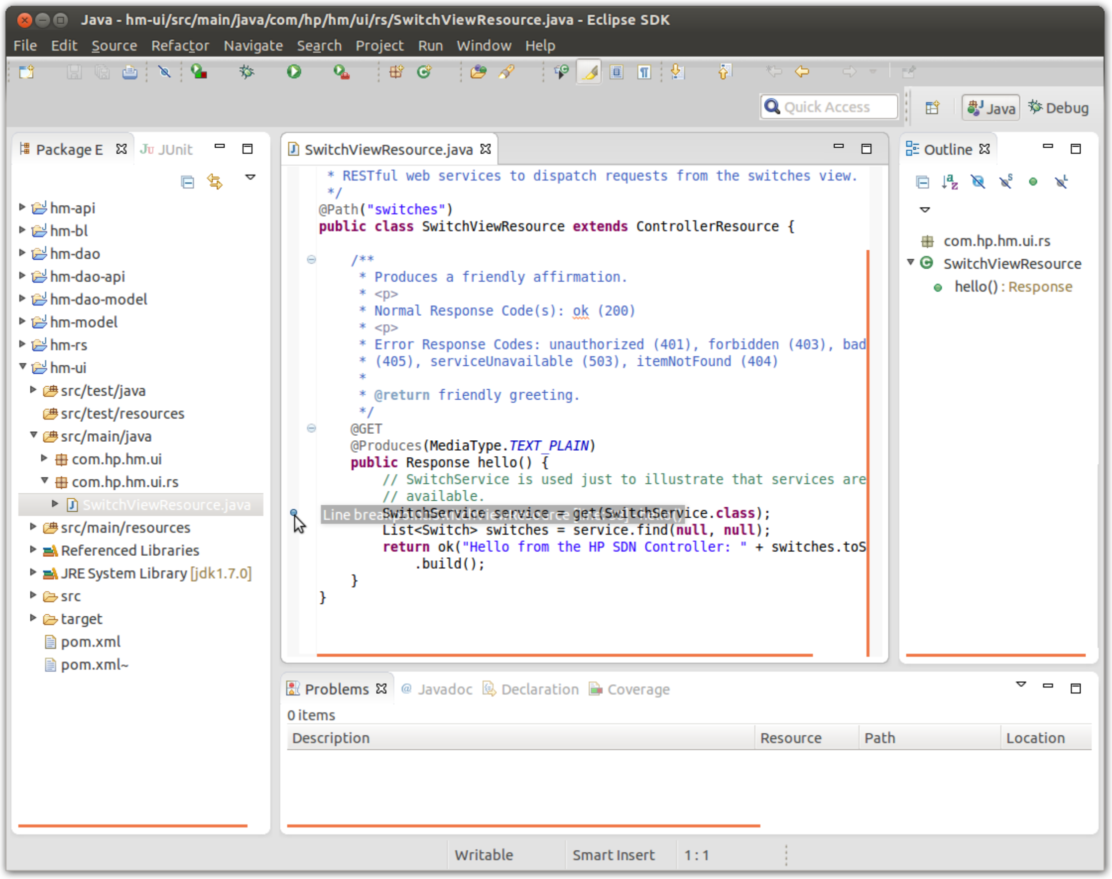
Port: 8787

☐ Allow termination of remote VM

Now add a break point and verify that the controller stops at such point. Skip the rest of this information if familiar with the Eclipse's debug perspective. Use the application developed at 4 Sample Application on page 106 at the point of section GUI-Specific REST API on page 173. 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 such 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, *SwitchViewResource* from module *hm-ui*, and add a break point in the only RESTful method there as illustrated in Figure 71.

Figure 71 Adding Break Point to SwitchViewResource.java



```
 * RESTful web services to dispatch requests from the switches view.
 */
@Path("switches")
public class SwitchViewResource extends ControllerResource {

    /**
     * Produces a friendly affirmation.
     * <p>
     * Normal Response Code(s): ok (200)
     * <p>
     * Error Response Codes: unauthorized (401), forbidden (403), bad
     * (405), serviceUnavailable (503), itemNotFound (404)
     *
     * @return friendly greeting.
     */
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public Response hello() {
        // SwitchService is used just to illustrate that services are
        // available.
        SwitchService service = get(SwitchService.class);
        List<Switch> switches = service.find(null, null);
        return ok("Hello from the HP SDN Controller: " + switches.toS
            .build();
    }
}
```

Figure 1

Now open the application and click on the *Refresh Data* button that displays a message retrieved from the REST API where the break point was just added. Figure 72 shows the sample application's view. After clicking *Refresh Data,* notice that a confirmation message from Eclipse requesting to change to the debug perspective, Figure 73. That means the controller hit the break point and now it can run step by step. Select *Yes* to continue.

Figure 72 Sample Application's View Remotely Debugged

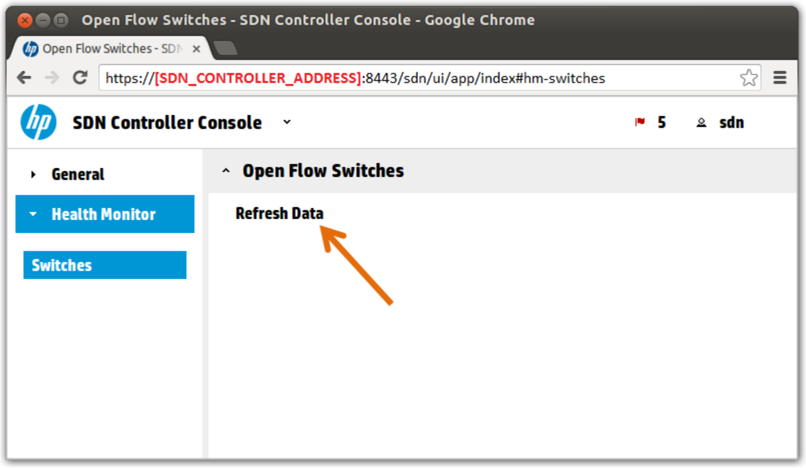
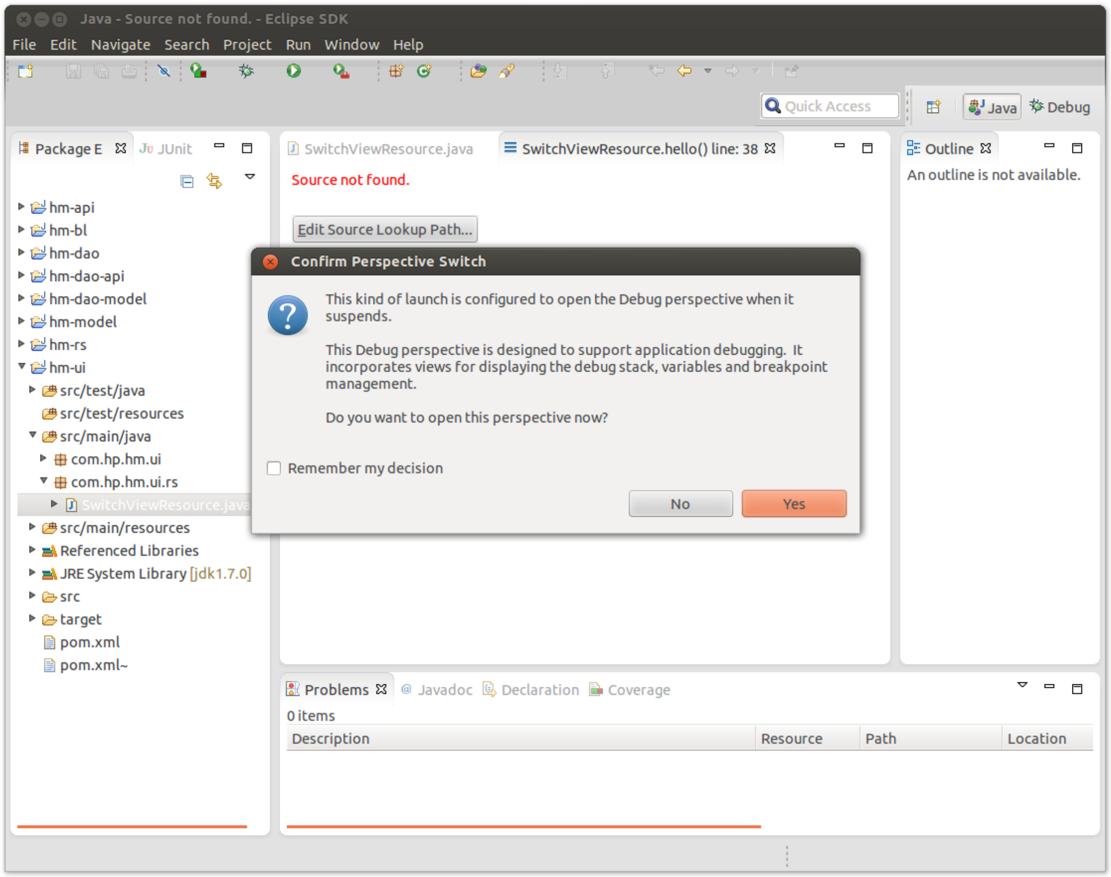

Open Flow Switches - SDN Controller Console - Google Chrome
https://[SDN_CONTROLLER_ADDRESS]:8443/sdn/ui/app/index#hm-switches
SDN Controller Console
5
sdn
General
Health Monitor
Switches
Open Flow Switches
Refresh Data

Figure 73 Perspective Switch



Source not found.

An outline is not available.

Confirm Perspective Switch

This kind of launch is configured to open the Debug perspective when it suspends.

This Debug perspective is designed to support application debugging. It incorporates views for displaying the debug stack, variables and breakpoint management.

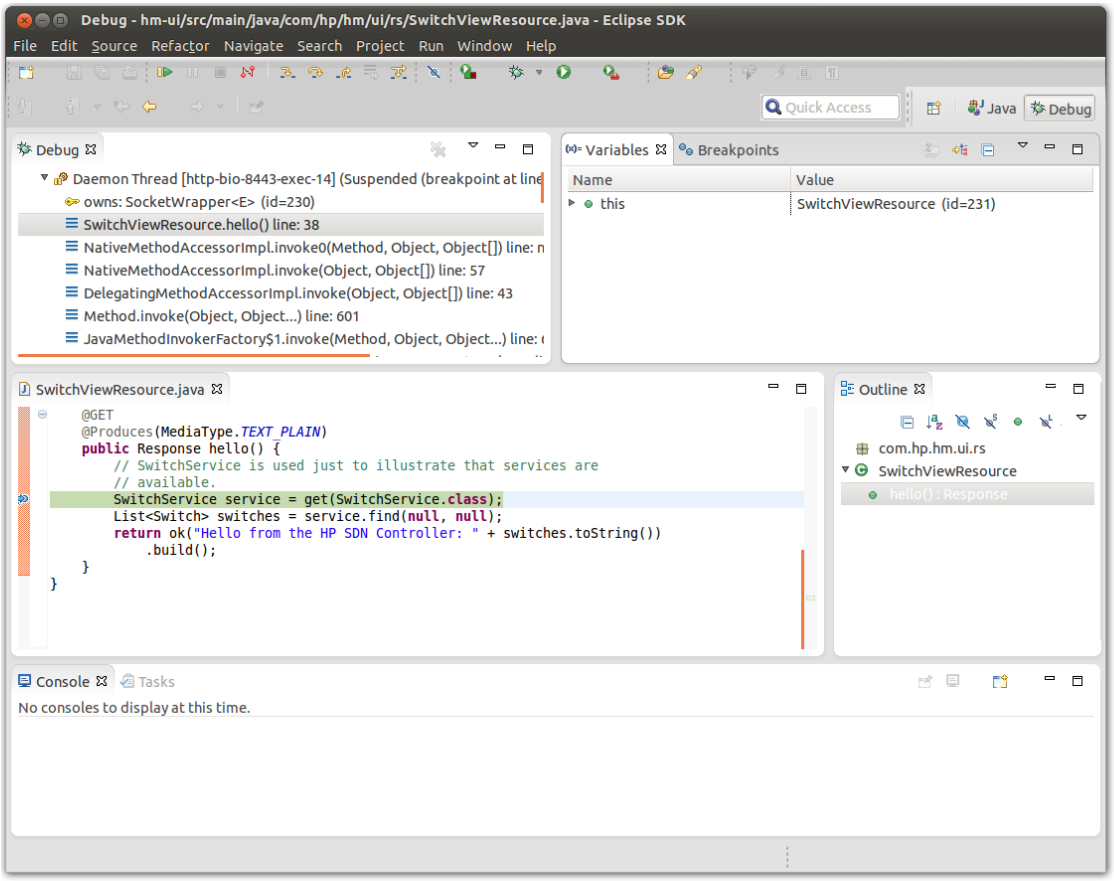
Do you want to open this perspective now?

Remember my decision

0 items

The code stopped at the break point can be seen, as in Figure 74, however, as we can see in Figure 73 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 213.

Figure 74 Debugging HP VAN SDN Controller



Debug - hm-ui/src/main/java/com/hp/hm/ui/rs/SwitchViewResource.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Java
Debug
Debug
Variables
Breakpoints
SwitchViewResource.java
Outline
Console
Tasks

```
Daemon Thread [http-bio-8443-exec-14] (Suspended (breakpoint at line
    owns: SocketWrapper<E> (id=230)
    SwitchViewResource.hello() line: 38
    NativeMethodAccessorImpl.invoke0(Method, Object, Object[]) line: n
    NativeMethodAccessorImpl.invoke(Object, Object[]) line: 57
    DelegatingMethodAccessorImpl.invoke(Object, Object[]) line: 43
    Method.invoke(Object, Object...) line: 601
    JavaMethodInvokerFactory$1.invoke(Method, Object, Object...) line:
```

Name	Value
this	SwitchViewResource (id=231)

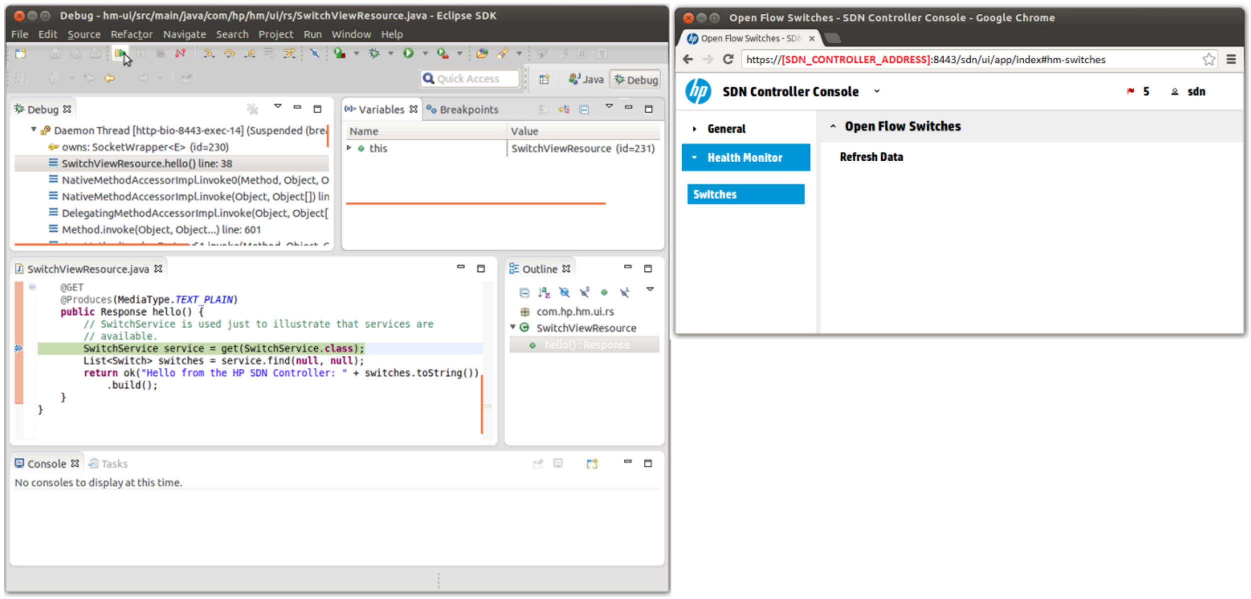
```
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public Response hello() {
        // SwitchService is used just to illustrate that services are
        // available.
        SwitchService service = get(SwitchService.class);
        List<Switch> switches = service.find(null, null);
        return ok("Hello from the HP SDN Controller: " + switches.toString())
            .build();
    }
}
```

```
com.hp.hm.ui.rs
SwitchViewResource
    hello() : Response
```

```
No consoles to display at this time.
```

Figure 75 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 76.

Figure 75 Controller's View Waiting for Code Being Debugged



Debug - hm-ui/src/main/java/com/hp/hm/ui/rs/SwitchViewResource.java - Eclipse SDK

```
Daemon Thread [http-bio-8443-exec-14] (Suspended (bre
    owns: SocketWrapper<E> (id=230)
    SwitchViewResource.hello() line: 38
    NativeMethodAccessorImpl.invoke0(Method, Object, O
    NativeMethodAccessorImpl.invoke(Object, Object[]) lin
    DelegatingMethodAccessorImpl.invoke(Object, Object[
    Method.invoke(Object, Object...) line: 601
```

Name	Value
this	SwitchViewResource (id=231)

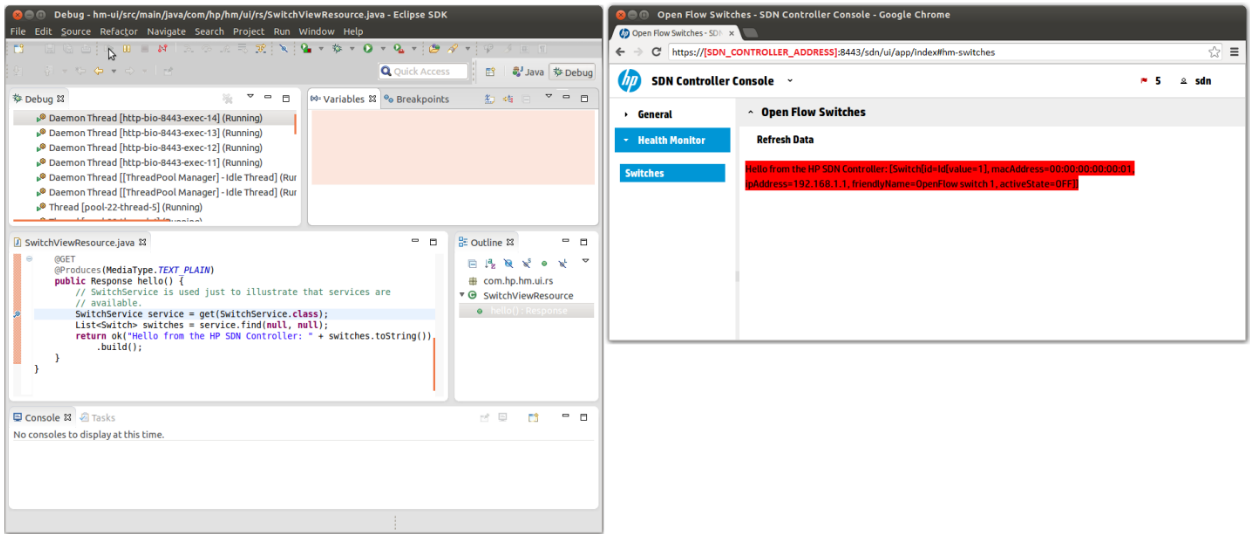
```java
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public Response hello() {
        // SwitchService is used just to illustrate that services are
        // available.
        SwitchService service = get(SwitchService.class);
        List<Switch> switches = service.find(null, null);
        return ok("Hello from the HP SDN Controller: " + switches.toString())
            .build();
    }
}
```

No consoles to display at this time.

Open Flow Switches - SDN Controller Console - Google Chrome

https://[SDN_CONTROLLER_ADDRESS]:8443/sdn/ui/app/index#hm-switches

Figure 76 SDN Controller's View Completed after Execution Resumed



Debug - hm-ui/src/main/java/com/hp/hm/ui/rs/SwitchViewResource.java - Eclipse SDK

```
Daemon Thread [http-bio-8443-exec-14] (Running)
Daemon Thread [http-bio-8443-exec-13] (Running)
Daemon Thread [http-bio-8443-exec-12] (Running)
Daemon Thread [http-bio-8443-exec-11] (Running)
Daemon Thread [[ThreadPool Manager] - Idle Thread] (Rur
Daemon Thread [[ThreadPool Manager] - Idle Thread] (Rur
Thread [pool-22-thread-5] (Running)
```

```java
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public Response hello() {
        // SwitchService is used just to illustrate that services are
        // available.
        SwitchService service = get(SwitchService.class);
        List<Switch> switches = service.find(null, null);
        return ok("Hello from the HP SDN Controller: " + switches.toString())
            .build();
    }
}
```

No consoles to display at this time.

Open Flow Switches - SDN Controller Console - Google Chrome

https://[SDN_CONTROLLER_ADDRESS]:8443/sdn/ui/app/index#hm-switches

```
Hello from the HP SDN Controller: [Switch[id=Id[value=1], macAddress=00:00:00:00:00:01,
ipAddress=192.168.1.1, friendlyName=OpenFlow switch 1, activeState=OFF]]
```

6 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

Device Node Manager

Node network service component is responsible for creating and maintaining the ARP Table. It also maintains the list of end points or end hosts. An ARP table is maintained for each VNET. The ARP table contains the MAC address for each host IP address present on the VNET. It does not contain router proxy ARP entries for IP addresses on remote networks. Orchestration software will configure a DHCP server or statically assign IP addresses to systems as they are deployed.

The ARP table sample data as shown in Table 7.

Table 7 ARP Table

IP Address	MAC	Vid	SwitchDpid	Port
10.250.100.1	00:af:cd:12:10:01	100	00:ae:c7:de:02:01:02:03	3
10.250.100.2	00:af:cd:12:10:20	110	00:ae:c7:de:02:01:02:03	4

Node Manager listens for PACKET_IN messages. Whenever PACKET_IN message is received by the Node Manager Subsystem of the controller, it uses the following methods in order to build its ARP and Node Table structures.

- It ignores the packet if the destination MAC Address is Multicast or broadcast.
- The connected port is identified by using Link Discovery/ Topology module.
- It extracts the source MAC Address from the Ethernet packet.
- It extracts the IP Address from the ARP or from DHCP reply packet. (The HP VAN SDN Controller 1.6 added IP learning to this list).

The Node Manager updates the existing ARP Tables using the IP Address as the key if it exists otherwise new entry is added to the corresponding table. Node Manager also provides certain configurable parameters which can be adjusted as per the network deployment:

- Enable or disable notifications for nodes added, removed or changed ("notify_listeners") : Node Manager provides a facility to send node event information to registered listeners. The nodes events are node add, delete and modify. However, the notification process is performance intensive, and therefore switched "off" by default. If the administrator wants to notify the registered listeners of the node events, this parameter can be turned "on".

Services published by Node Manager Service

Node Manager Network component publishes following API via OSGI declarative services and REST API. The published API can be broadly categorized in two sub categories:

- IP Address Control functionality

- Network Node state tracking functionality

The API's are described in the following listing. See Javadoc on page 9 for details.

NodeService:

```
public interface NodeService {

    NetworkNode getNetworkNode(MacAddress mac, VId vid);

    NetworkNode getNetworkNode(IpAddress ip, VId vid);

    Set<NetworkNode> getNetworkNodes(VId vid);

    MacAddress getMacAddress(IpAddress ip, VId vid);

    void clearNodeTable(VId vid);

    void clearNodeTable(Set<VId> vids);

    void clearNodeTable();

    ARPTable getArpTable(VId vid);

    Set<VId> getVnetIDs();

    void addNetworkNodeListener(NetworkNodeListener nnl);

    void removeNetworkNodeListener(NetworkNodeListener nnl);
}
```

Link Discovery

This Network services layer is intended to build up the knowledge about 'links' between the network elements residing in the controller's domain. This intelligence about the linkages between the network elements would serve as base for other apps like Routing, Topology etc, and enable the overall orchestration of virtual networks.

The information is built by way of listening to device connected/disconnected events from the controller, as well as by way of inducing the network elements to send LLDP TLV's. Details about this would be captured in subsequent sections.

The Link table sample data is shown in Table 8.

Table 8 Link Table

Source Device	Source Port	Destination Device	Destination Port
03:e7:00:26:f1:29:af:00	1	03:e7:00:23:47:ba:05:40	23
03:e8:00:23:47:ba:05:40	11	03:e8:00:26:f1:29:af:00	8

Device Interaction

- This application listens to LLDP messages, Link up/down messages.
- The application sends out LLDP PDUs to all devices.

Services provided by Link Manager

- Learns and maintains all inter switch links in the control domain.
- Used by the Topology Module to construct end-to-end paths.
- The applications can use link discovery services to reconfigure flows when link goes down.
- Users can avoid sending LLDP discover packet on certain ports like edge port, by adding ports to the "Suppressed LLDP Ports"

By way of API's consumable by other peer Network services as well as by elements to its north in the solution stack, the Link Discovery Network Service essentially provides the key services captured below.

Key Terms

- InfraConnectionPoint: Port participating in intra switch physical links.
- Suppress LLDP Ports: To avoid sending LLDP discover packet on certain ports like edge port, the Link Discovery module maintains a special list of ports which are called as "Suppressed LLDP Ports"

Get all discovered physical links in the network

```
Set<Link> getAllLinks();
```

Get all physical links for device

```
Set<Link> getDeviceLinks(DataPathId dpid);
```

Check if a given connection point is a infrastructure connection point

```
boolean isInfraPort(ConnectionPoint cp);
```

Get all ports on which LLDP is suppressed

```
Set<ConnectionPoint> getSuppressLLDPsInfo();
```

Add switch port to LLDP suppressed list

```
void addToSuppressLLDPs(DataPathId dpid, BigPortNumber port);
```

Remove a switch port from the LLDP suppressed list

```
void removeFromSuppressLLDPs(DataPathId dpid, BigPortNumber port);
```

Register for getting the link discovery events

```
void addListener(LinkListener listener);
```

Unregister from getting the link discovery events

```
void removeListener(LinkListener listener);
```

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 tress 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 {s1}, can provide the list of ports that this service has discovered
- For two switches {s1, s2}, the service can indicate if they are "strongly connected" i.e form part of same cluster.
- For a given {switch, port} pair, it would indicate if they participate as a "connection point" (if they form an 'edge port')
- For a given {switch, port} pair, this service can indicate if ingress broadcast is allowed through 'port'

  - 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

The API's are described in the following listing. See Javadoc on page 9 for details.

Topology Service:

```
public interface TopologyService {

    boolean pathExists(DataPathId source, DataPathId dest);

    L2Path path(DataPathId source, DataPathId dest);

    Set<BigPortNumber> ports(DataPathId switchDpid);

    boolean isConnectionPoint(DataPathId switchDpid,
        BigPortNumber portId);

    List<TopologyCluster> clusters();

    TopologyCluster cluster(DataPathId switchDpid);

    boolean participateInBroadcast(DataPathId switchDpid,
        BigPortNumber portId);

    Map<DataPathId, Link> tree(long clusterId);

    void addListener(TopologyListener listener);

    void removeListener(TopologyListener listener);
}
```

Appendix A

Eclipse

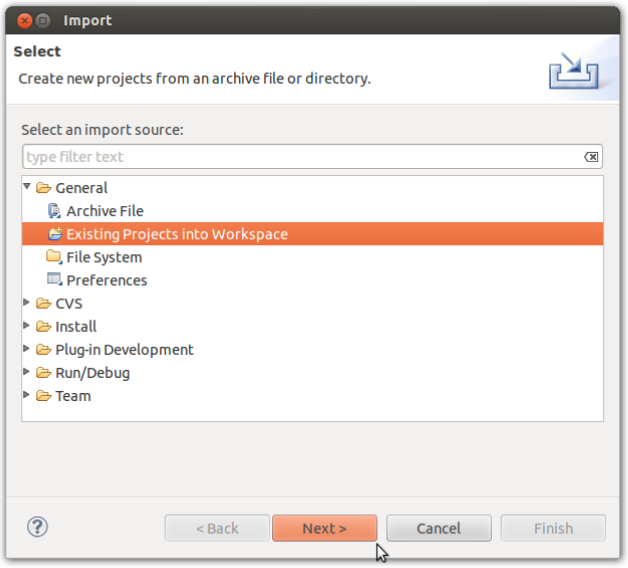
This appendix describes some of the Eclipse [47] features that an SDN Controller Application developer will often use.

Importing Java Projects

To import an entire Eclipse project from an archive file, follow these steps:

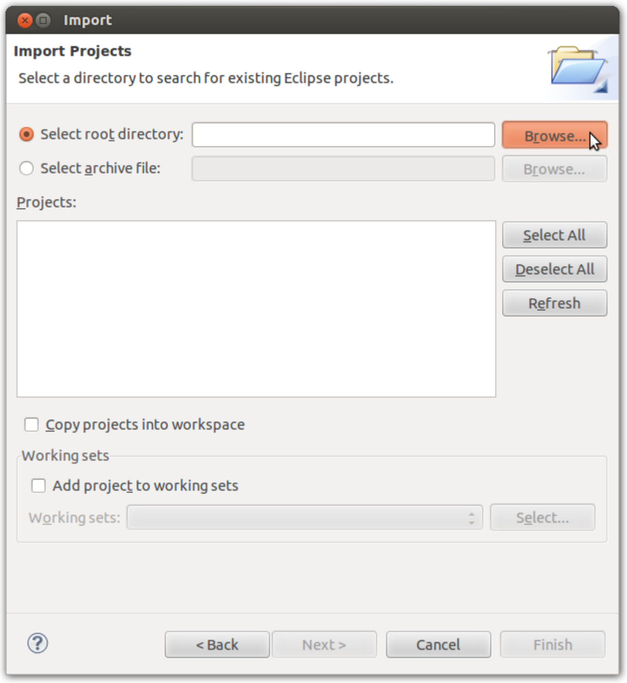
1. Go to *File* → *Import*. The following dialog appears.

Figure 77 Eclipse Source Selection Dialog (Import Java Project)



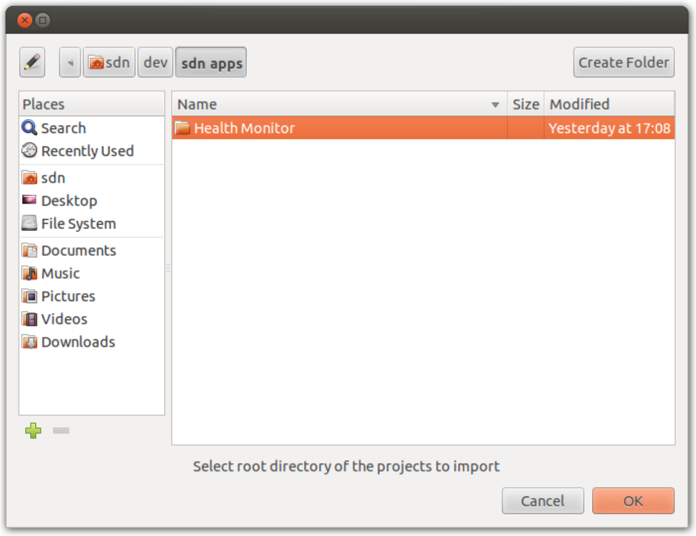
2. Select *Existing Projects into Workspace*. Then Click the button *next*.

Figure 78 Eclipse Directory Selection Dialog (Import Java Project)



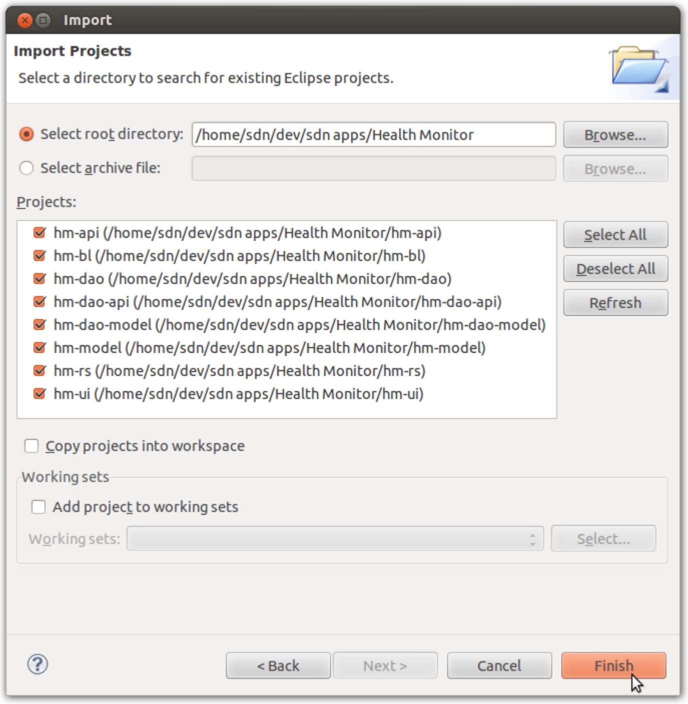
3. Click ***Browse*** button and find the root folder (SDN Controller Application Workspace folder) on your hard disk. Several projects can be imported together depending on the selected root directory. Then click *OK* to select it.

Figure 79 Eclipse File Chooser Dialog (Import Java Project)



4. Click *Finish* to perform the import.

Figure 80 Import Dialog (Import java Project)



Import Projects

Select a directory to search for existing Eclipse projects.

(●) Select root directory: /home/sdn/dev/sdn apps/Health Monitor [Browse...]

( ) Select archive file: [Browse...]

Projects:

- ☑ hm-api (/home/sdn/dev/sdn apps/Health Monitor/hm-api)
- ☑ hm-bl (/home/sdn/dev/sdn apps/Health Monitor/hm-bl)
- ☑ hm-dao (/home/sdn/dev/sdn apps/Health Monitor/hm-dao)
- ☑ hm-dao-api (/home/sdn/dev/sdn apps/Health Monitor/hm-dao-api)
- ☑ hm-dao-model (/home/sdn/dev/sdn apps/Health Monitor/hm-dao-model)
- ☑ hm-model (/home/sdn/dev/sdn apps/Health Monitor/hm-model)
- ☑ hm-rs (/home/sdn/dev/sdn apps/Health Monitor/hm-rs)
- ☑ hm-ui (/home/sdn/dev/sdn apps/Health Monitor/hm-ui)

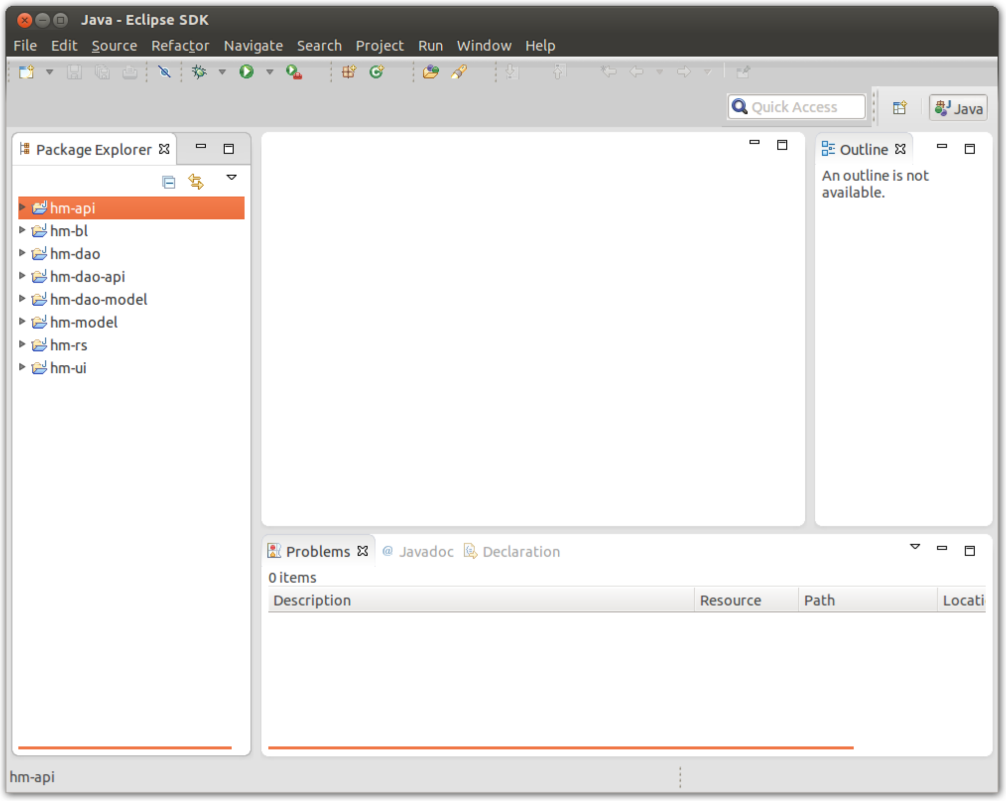
☐ Copy projects into workspace

Working sets

☐ Add project to working sets

Working sets:

Figure 81 Eclipse Imported Projects



An outline is not available.

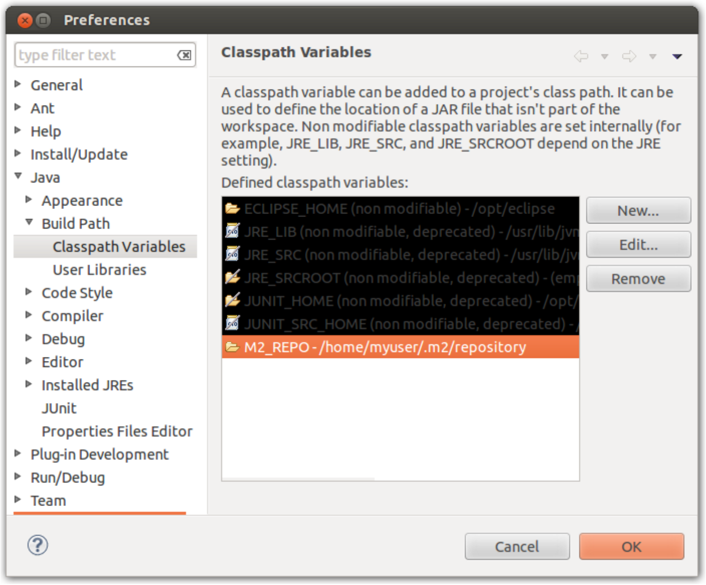
0 items

Description	Resource	Path	Locati

Setting M2_REPO Classpath Variable

Go to *Window* → *Preferences*. Then add the location of Maven repository as illustrated in Figure 82.

Figure 82 Setting M2_REPO Classpath Variable



Classpath Variables

A classpath variable can be added to a project's class path. It can be used to define the location of a JAR file that isn't part of the workspace. Non modifiable classpath variables are set internally (for example, JRE_LIB, JRE_SRC, and JRE_SRCROOT depend on the JRE setting).

Defined classpath variables:

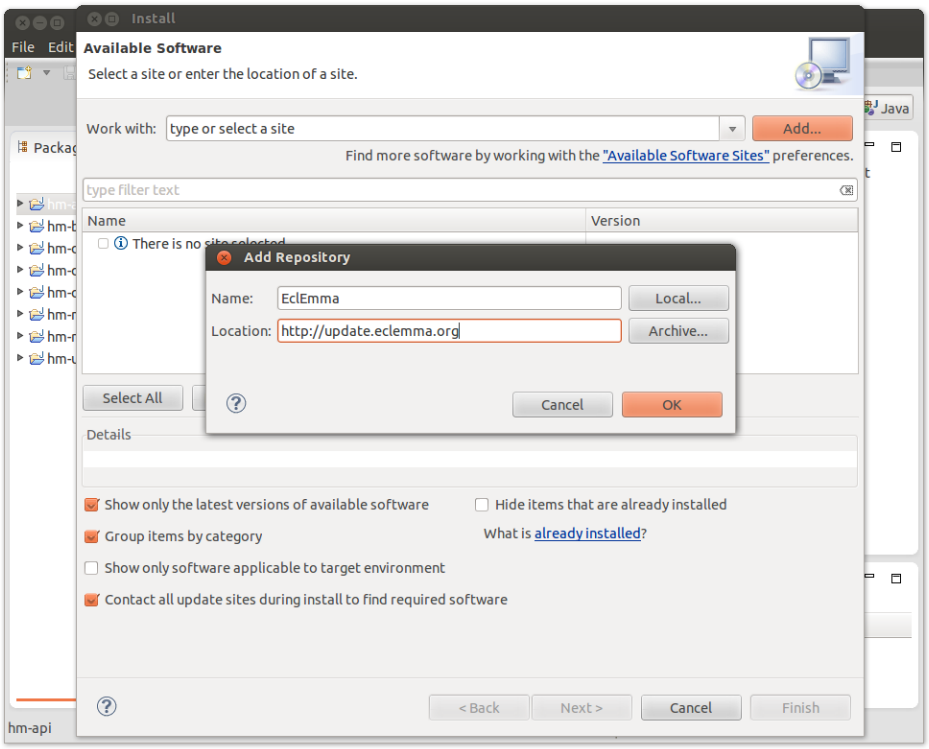
```
ECLIPSE_HOME (non modifiable) - /opt/eclipse
JRE_LIB (non modifiable, deprecated) - /usr/lib/jvı
JRE_SRC (non modifiable, deprecated) - /usr/lib/jv
JRE_SRCROOT (non modifiable, deprecated) - (em
JUNIT_HOME (non modifiable, deprecated) - /opt/
JUNIT_SRC_HOME (non modifiable, deprecated) -
M2_REPO - /home/myuser/.m2/repository
```

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 83. Use the URL from step 1 as the location.

Figure 83 Adding Plug-in Repository


Install

Available Software

Select a site or enter the location of a site.

Work with: type or select a site

Find more software by working with the "Available Software Sites" preferences.

Add Repository

Name: EclEmma

Location: http://update.eclemma.org

Show only the latest versions of available software

Group items by category

Show only software applicable to target environment

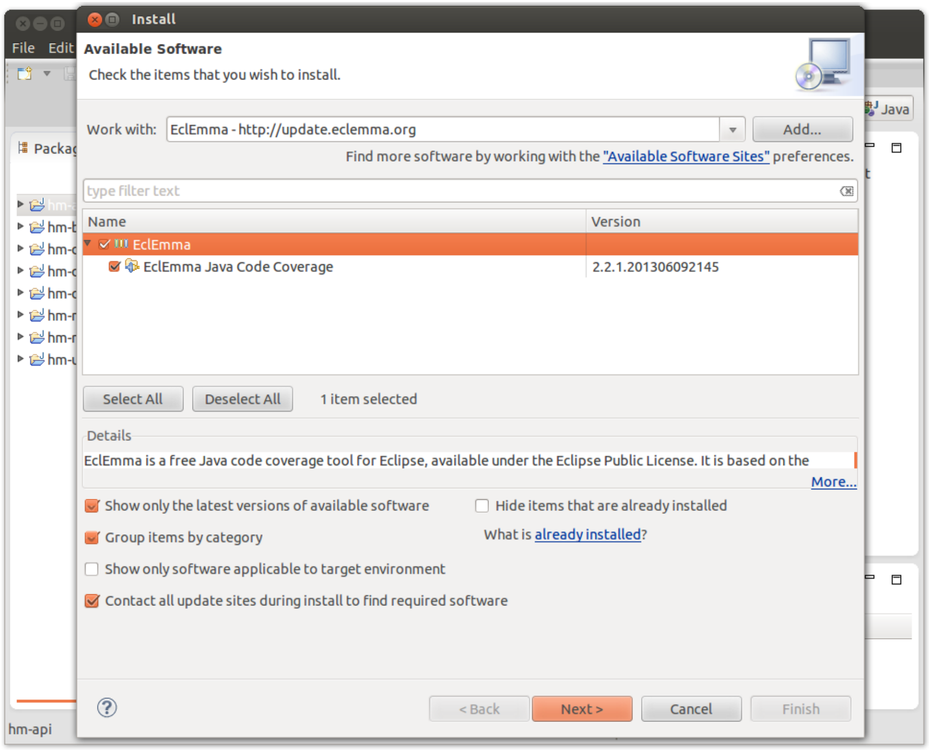
Contact all update sites during install to find required software

Hide items that are already installed

What is already installed?

3. Select the checkbox of the plug-in and follow the installation wizard.

Figure 84 Eclipse's Plug-in Installation Wizard


Install

Available Software

Check the items that you wish to install.

Work with: EclEmma - http://update.eclemma.org

Find more software by working with the "Available Software Sites" preferences.

Name	Version
EclEmma	
EclEmma Java Code Coverage	2.2.1.201306092145

1 item selected

Details

EclEmma is a free Java code coverage tool for Eclipse, available under the Eclipse Public License. It is based on the

Show only the latest versions of available software

Group items by category

Show only software applicable to target environment

Contact all update sites during install to find required software

Hide items that are already installed

What is already installed?

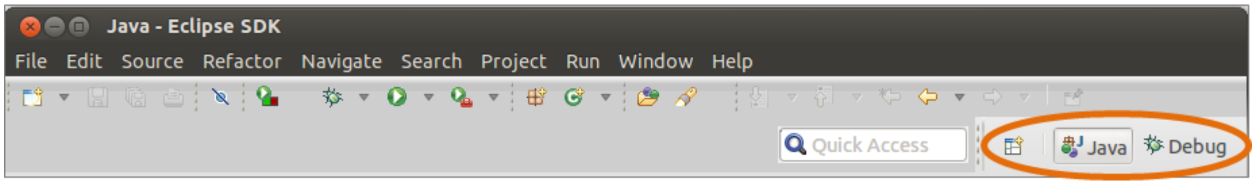
Eclipse Perspectives

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 85.

Figure 85 Perspectives Tool Tar

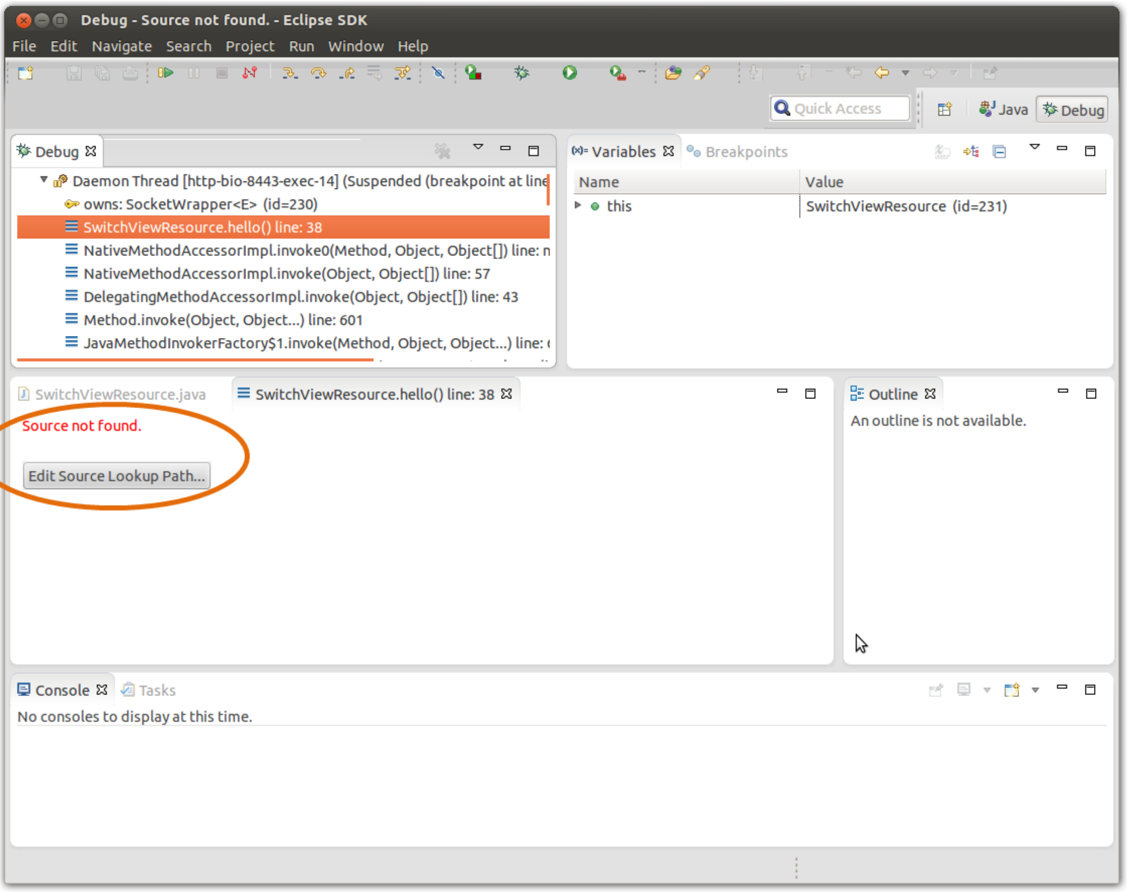

Java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Java
Debug

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 86 (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 86 to open the Edit Source Lookup Path dialog.

Figure 86 Source Not Found


Debug - Source not found. - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Quick Access
Java
Debug
Debug
Variables
Breakpoints
SwitchViewResource.java
Outline
Console
Tasks
Edit Source Lookup Path...

```
Daemon Thread [http-bio-8443-exec-14] (Suspended (breakpoint at line
  owns: SocketWrapper<E> (id=230)
  SwitchViewResource.hello() line: 38
  NativeMethodAccessorImpl.invoke0(Method, Object, Object[]) line: n
  NativeMethodAccessorImpl.invoke(Object, Object[]) line: 57
  DelegatingMethodAccessorImpl.invoke(Object, Object[]) line: 43
  Method.invoke(Object, Object...) line: 601
  JavaMethodInvokerFactory$1.invoke(Method, Object, Object...) line:
```

Name	Value
this	SwitchViewResource (id=231)

SwitchViewResource.hello() line: 38

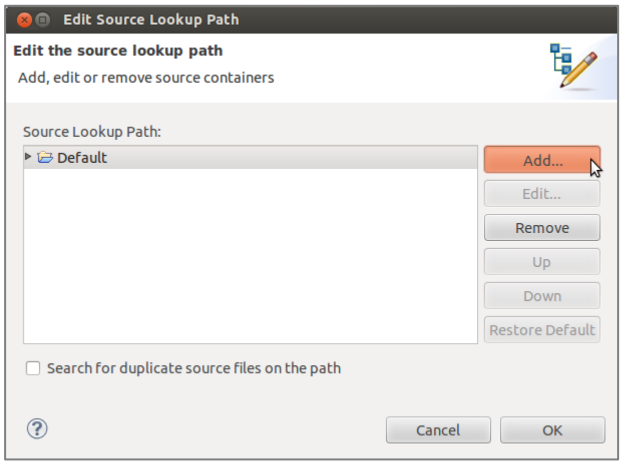
Source not found.

An outline is not available.

No consoles to display at this time.

2. Click *Add* button from the Edit Source Lookup Path dialog.

Figure 87 Edit Source Lookup Path Dialog



Edit the source lookup path

Add, edit or remove source containers

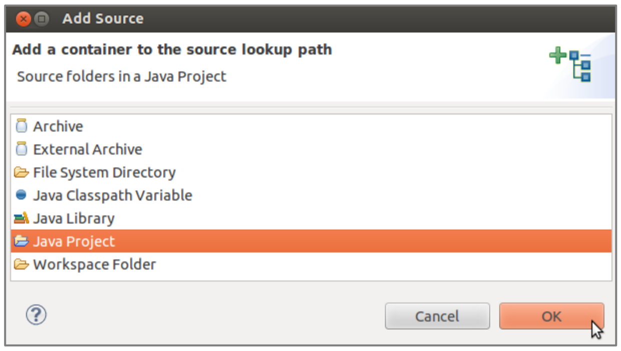
Source Lookup Path:

Default

☐ Search for duplicate source files on the path

3. Select Java Project as the source.

Figure 88 Lookup Path Resource Type


Add Source

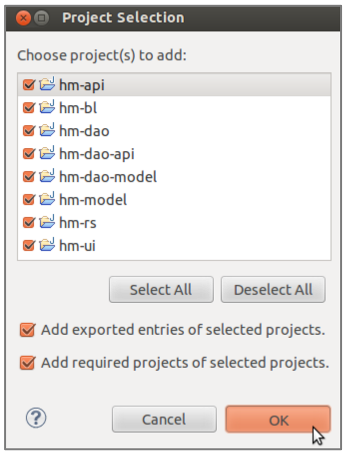
Add a container to the source lookup path

Source folders in a Java Project

- Archive
- External Archive
- File System Directory
- Java Classpath Variable
- Java Library
- Java Project
- Workspace Folder

4. Select projects.

Figure 89 Lookup Path Resource Selections


Project Selection

Choose project(s) to add:

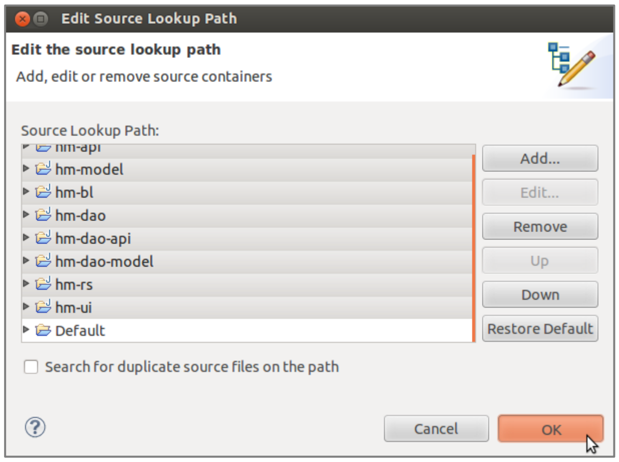
- [✓] hm-api
- [✓] hm-bl
- [✓] hm-dao
- [✓] hm-dao-api
- [✓] hm-dao-model
- [✓] hm-model
- [✓] hm-rs
- [✓] hm-ui

[✓] Add exported entries of selected projects.

[✓] Add required projects of selected projects.

5. Confirm configuration.

Figure 90 Source Lookup Path Confirmation


Edit Source Lookup Path

Edit the source lookup path

Add, edit or remove source containers

Source Lookup Path:

- hm-api
- hm-model
- hm-bl
- hm-dao
- hm-dao-api
- hm-dao-model
- hm-rs
- hm-ui
- Default

[ ] Search for duplicate source files on the path

Appendix B

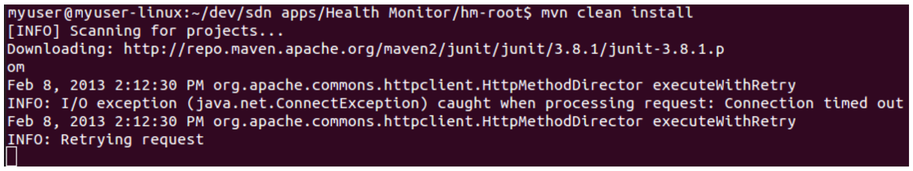
Troubleshooting

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.

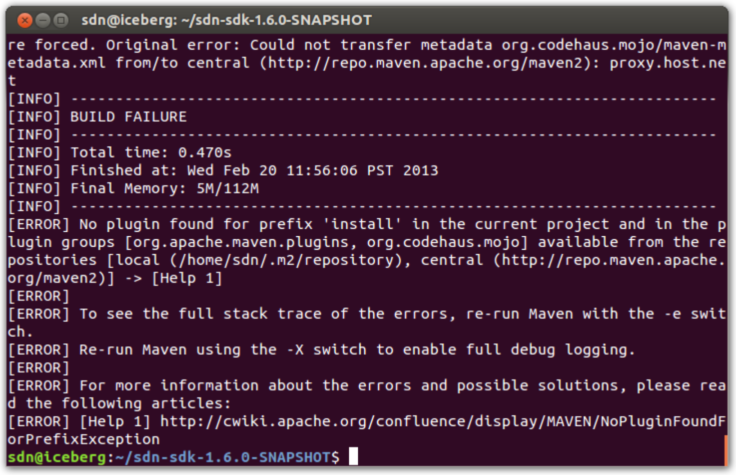
Figure 91 Maven Problem: No Proxy Configured



```
myuser@myuser-linux:~/dev/sdn apps/Health Monitor/hm-root$ mvn clean install
[INFO] Scanning for projects...
Downloading: http://repo.maven.apache.org/maven2/junit/junit/3.8.1/junit-3.8.1.p
om
Feb 8, 2013 2:12:30 PM org.apache.commons.httpclient.HttpMethodDirector executeWithRetry
INFO: I/O exception (java.net.ConnectException) caught when processing request: Connection timed out
Feb 8, 2013 2:12:30 PM org.apache.commons.httpclient.HttpMethodDirector executeWithRetry
INFO: Retrying request
```

The output shown in Figure 92 is also related to the proxy problem and it happens when Maven proxy configuration is incorrect.

Figure 92 Maven Problem: Invalid Proxy Configuration



```
sdn@iceberg: ~/sdn-sdk-1.6.0-SNAPSHOT
re forced. Original error: Could not transfer metadata org.codehaus.mojo/maven-m
etadata.xml from/to central (http://repo.maven.apache.org/maven2): proxy.host.ne
t
[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 0.470s
[INFO] Finished at: Wed Feb 20 11:56:06 PST 2013
[INFO] Final Memory: 5M/112M
[INFO] ------------------------------------------------------------------------
[ERROR] No plugin found for prefix 'install' in the current project and in the p
lugin groups [org.apache.maven.plugins, org.codehaus.mojo] available from the re
positories [local (/home/sdn/.m2/repository), central (http://repo.maven.apache.
org/maven2)] -> [Help 1]
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e swit
ch.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
[ERROR]
[ERROR] For more information about the errors and possible solutions, please rea
d the following articles:
[ERROR] [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/NoPluginFoundF
orPrefixException
sdn@iceberg:~/sdn-sdk-1.6.0-SNAPSHOT$
```

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:

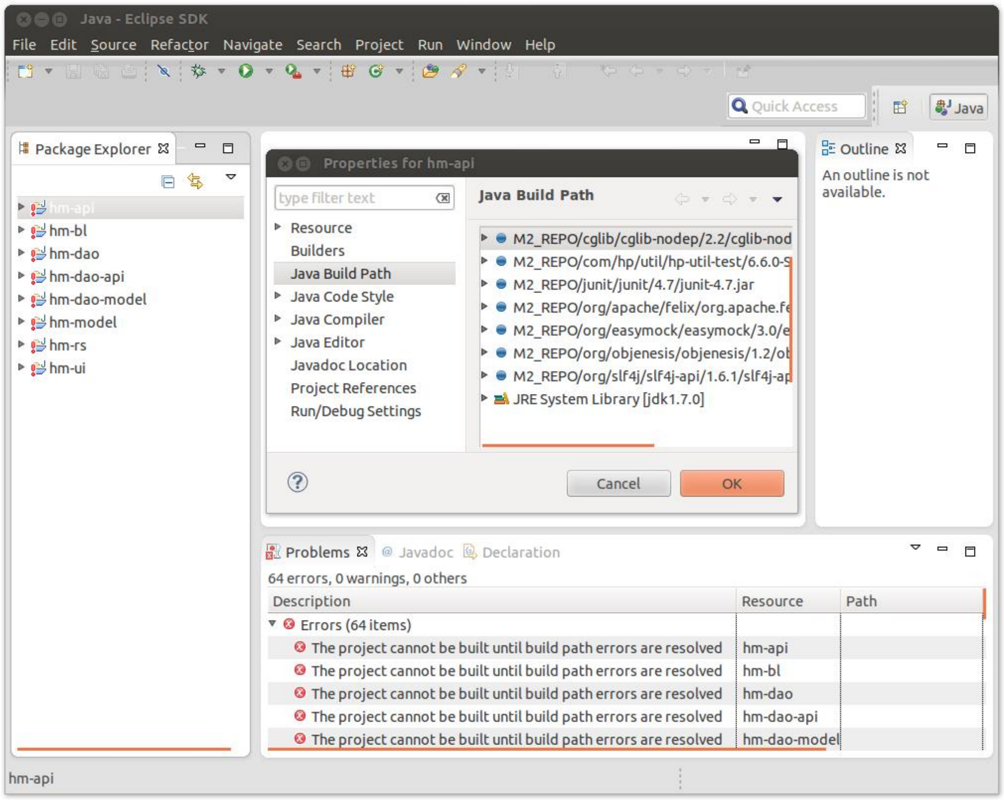
```
<proxies>
  <proxy>
    <id>optional</id>
    <active>true</active>
    <protocol>http</protocol>
    <username>proxyuser</username>
    <password>proxypass</password>
    <host>web-proxy.rose.hp.com</host>
    <port>8088</port>
    <nonProxyHosts>local.net|some.host.com</nonProxyHosts>
  </proxy>
</proxies>
```

Path Errors in Eclipse Projects after Importing

Problem

This problem occurs when the M2_REPO variable is not set in Eclipse.

Figure 93 Eclipse Missing M2_REPO Configuration Problem



M2_REPO/cglib/cglib-nodep/2.2/cglib-nod
M2_REPO/com/hp/util/hp-util-test/6.6.0-S
M2_REPO/junit/junit/4.7/junit-4.7.jar
M2_REPO/org/apache/felix/org.apache.fe
M2_REPO/org/easymock/easymock/3.0/e
M2_REPO/org/objenesis/objenesis/1.2/ob
M2_REPO/org/slf4j/slf4j-api/1.6.1/slf4j-ap
JRE System Library [jdk1.7.0]

64 errors, 0 warnings, 0 others

Description	Resource	Path
Errors (64 items)		
The project cannot be built until build path errors are resolved	hm-api	
The project cannot be built until build path errors are resolved	hm-bl	
The project cannot be built until build path errors are resolved	hm-dao	
The project cannot be built until build path errors are resolved	hm-dao-api	
The project cannot be built until build path errors are resolved	hm-dao-model	

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 211.

Bibliography

[1] Hewlett-Packard, "REST Guidelines," [Online]. Available: http://h17007.www1.hp.com/us/en/networking/solutions/technology/sdn/devcenter/index.aspx.

[2] Jersey, "Jersey," [Online]. Available: https://jersey.java.net/.

[3] Oracle, "Servlets," [Online]. Available: http://www.oracle.com/technetwork/java/index-jsp-135475.html.

[4] B. Basham, K. Sierra and B. Bates, Head First Servlets & JSP, O'REILLY, 2008.

[5] OSGi Alliance, "OSGi," [Online]. Available: http://www.osgi.org/Main/HomePage.

[6] T. E. Foundation, "Equinox," [Online]. Available: http://www.eclipse.org/equinox/.

[7] T. E. Foundation, "VIRGO," [Online]. Available: http://www.eclipse.org/virgo/.

[8] T. A. S. Foundation, "Tomcat," [Online]. Available: http://tomcat.apache.org/.

[9] OpenStack, "Keystone," [Online]. Available: https://wiki.openstack.org/wiki/Keystone.

[10] The Apache Software Foundation, "ZooKeeper," [Online]. Available: http://zookeeper.apache.org/.

[11] The Apache Software Foundation, "Cassandra," [Online]. Available: http://cassandra.apache.org/.

[12] OpenFlow, "OpenFlow," [Online]. Available: http://www.openflow.org/.

[13] O. N. Foundation, "Open Networking Foundation," [Online]. Available: https://www.opennetworking.org/.

[14] I. VMware, "VMware," [Online]. Available: http://www.vmware.com/.

[15] Hewlett-Packard, "HP VAN SDN Controller Installation Guide," [Online]. Available: www.hp.com/support/manuals.

[16] Oracle, "Java," [Online]. Available: http://www.oracle.com/technetwork/java/index.html.

[17] The Apache Software Foundation, "Maven," [Online]. Available: http://maven.apache.org/.

[18] Haxx, "Curl," [Online]. Available: http://curl.haxx.se/.

[19] Hewlett-Packard, "HP SDN Developer Kit," [Online]. Available: http://h17007.www1.hp.com/us/en/networking/solutions/technology/sdn/devcenter/index.aspx.

[20] Wikipedia, "Plain Old Java Object (POJO)," [Online]. Available: http://en.wikipedia.org/wiki/Plain_Old_Java_Object.

[21] IBM, "RESTful Web Services," [Online]. Available: http://www.ibm.com/developerworks/webservices/library/ws-restful/.

[22] Oracle, "Javadoc," [Online]. Available: http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html.

[23] OSGi Alliance, "OSGi Services - Configuration Admin," [Online]. Available: http://www.osgi.org/javadoc/r4v42/org/osgi/service/cm/ConfigurationAdmin.html.

[24] The Apache Software Foundation, "Configuration Admin Service," [Online]. Available: http://felix.apache.org/documentation/subprojects/apache-felix-config-admin.html.

[25] OSGi Alliance, "OSGi Services - MetaType," [Online]. Available: http://www.osgi.org/javadoc/r4v42/org/osgi/service/metatype/package-summary.html.

[26] The Apache Software Foundation, "Metatype Service," [Online]. Available: http://felix.apache.org/site/apache-felix-metatype-service.html.

[27] The Apache Software Foundation, "Maven Plug-ins - SCR Annotations," [Online]. Available: http://felix.apache.org/documentation/subprojects/apache-felix-maven-scr-plugin/scr-annotations.html.

[28] The Apache Software Foundation, "Maven Plug-ins - SCR Plugin," [Online]. Available: http://felix.apache.org/documentation/subprojects/apache-felix-maven-scr-plugin.html.

[29] Open Networking Foundation, "OpenFlow Specification," [Online]. Available: https://www.opennetworking.org/sdn-resources/onf-specifications/openflow.

[30] Hewlett-Packard, "HP VAN SDN Controller Admin Guide," [Online]. Available: www.hp.com/support/manuals.

[31] Oracle, "DAO Pattern," [Online]. Available: http://www.oracle.com/technetwork/java/dataaccessobject-138824.html.

[32] Wikipedia, "Data Transfer Object Pattern," [Online]. Available: http://en.wikipedia.org/wiki/Data_transfer_object.

[33] Wikipedia, "JavaBeans," [Online]. Available: http://en.wikipedia.org/wiki/JavaBeans.

[34] R. S. Hall, K. Pauls, S. McCulloch and D. Savage, OSGi in Action - Creating

Modular Applications in Java, Manning Publications Co., 2011.

[35] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns Elements of Reusable Object-Oriented Software, Addison Wesley, 2007.

[36] Oracle, "The Java EE 5 Tutorial," [Online]. Available: http://docs.oracle.com/javaee/5/tutorial/doc/.

[37] JSON, "JSON," [Online]. Available: http://www.json.org/.

[38] SourceForge, "EasyMock," [Online]. Available: http://www.easymock.org/.

[39] Codehaus, "Jackson Java JSON-processor," [Online]. Available: http://jackson.codehaus.org/.

[40] D. Flanagan, JavaScript The definiteve Guide, O'REILLY, 2011.

[41] Wikipedia, "Cascading Style Sheets," [Online]. Available: https://en.wikipedia.org/wiki/Cascading_Style_Sheets.

[42] T. j. Foundation, "jQuery," [Online]. Available: http://jquery.com/.

[43] Wikipedia, "AJAX," [Online]. Available: http://en.wikipedia.org/wiki/Ajax_(programming).

[44] QOS.ch., "SLF4J," [Online]. Available: http://www.slf4j.org/.

[45] Mininet Team, "Mininet," [Online]. Available: http://mininet.org/.

[46] M. G. &. C. K. a. Contributors, "EclEmma," [Online]. Available: http://www.eclemma.org/.

[47] The Eclipse Foundation, "Eclipse," [Online]. Available: http://www.eclipse.org/.

[48] Hewlett-Packard, "HP SKI," [Online]. Available: https://genesis.americas.hpqcorp.net/redmine/projects/hp-util/wiki/Ski-TOC.

[49] W3C, "W3C," [Online]. Available: http://www.w3.org/.