HP OpenView Service Desk

Web API Programmer's Guide

Software Version: 5.10

For the Windows and UNIX Operating Systems

Legal Notices

Warranty.

The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

The information contained herein is subject to change without notice.

Restricted Rights Legend.

Confidential computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Copyright Notices.

©Copyright 2005-2006 Hewlett-Packard Development Company, L.P.

Trademark Notices.

Adobe® and Acrobat® are trademarks of Adobe Systems Incorporated.

HP-UX Release 10.20 and later and HP-UX Release 11.00 and later (in both 32 and 64-bit configurations) on all HP 9000 computers are Open Group UNIX 95 branded products.

Java™ and all Java based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Oracle® is a registered U.S. trademark of Oracle Corporation, Redwood City, California.

UNIX® is a registered trademark of the Open Group.

Microsoft®, Windows NT® and Windows XP® are U.S. registered trademark of Microsoft Corporation.

Contents

Contents

Contents

Contents

Documentation Updates

This manual's title page contains the following identifying information:

- Version number, which indicates the software version.

- Document release date, which changes each time the document is updated.

- Software release date, which indicates the release date of this version of the software.

To check for recent updates or to verify that you are using the most recent edition, visit the following URL:

http://ovweb.external.hp.com/lpe/doc_serv/

You will also receive updated or new editions if you subscribe to the appropriate product support service. Contact your HP sales representative for details.

NOTE   For an overview of the latest information on supported platforms, installation prerequisites, software deliverables, and so on, refer to the Service Desk 5.10 Release Notes.

Table 1   Changes to this Document

Chapter	Changes
Documentation Updates	Added note on Service Desk 5.10 Release Notes.

Support

Please visit the HP OpenView support web site at:

http://www.hp.com/managementsoftware/support

This web site provides contact information and details about the products, services, and support that HP OpenView offers.

HP OpenView online software support provides customer self-solve capabilities. It provides a fast and efficient way to access interactive technical support tools needed to manage your business. As a valuable support customer, you can benefit by using the support site to:

- Search for knowledge documents of interest
- Submit enhancement requests online
- Download software patches
- Submit and track progress on support cases
- Manage a support contract
- Look up HP support contacts
- Review information about available services
- Enter discussions with other software customers
- Research and register for software training

Most of the support areas require that you register as an HP Passport user and log in. Many also require a support contract.

To find more information about access levels, go to:

http://www.hp.com/managementsoftware/access_level

To register for an HP Passport ID, go to:

http://www.managementsoftware.hp.com/passport-registration.html

Preface

This document describes procedures for using the Service Desk Web API. This Application Program Interface (API) enables you to develop your own web applications around HP OpenView Service Desk, integrate Service Desk into local applications, and add custom functionality to your Service Desk implementation.

This guide provides information about the architecture and use of the Web API, and is intended for anyone developing applications using the Web API. It is assumed that you have an administrator-level knowledge of Service Desk, and a reasonable knowledge of Java.

This guide is organized as follows:

- Chapter 1, "Getting Started," on page 13 explains how to install the Web API.

- Chapter 2, "Programming Interface," on page 31 describes the structure of the API.

- Chapter 3, "Changes From Service Desk 4.5 to 5.0," on page 69 explains what has changed since the Service Desk 4.5 version.

1 Getting Started

This chapter introduces the HP OpenView Service Desk Web API. It describes how to install the Web API and use it to build applications.

Web API

The Service Desk Web API is the Sun Java programming interface to the HP OpenView Service Desk application. It opens up the Service Desk application, its object model, and its data to your specific requirements.

The Web API enables you to develop web applications around the Service Desk core. In addition, it can be used to integrate Service Desk with local applications, and to add custom functionality to your Service Desk implementation.

The Web API allows you to customize how processes are implemented using Service Desk. You can use the Web API to define alternative assignment algorithms or to automatically close items if related items have been closed by using commands that are called from Service Desk smart actions or from database rule actions, for example.

NOTE   To compile and run the examples provided with the Web API you need the Sun Java Development Environment (JDE) and a working Service Desk installation. See the *HP OpenView Service Desk Installation Guide* for more information.

Service Desk Architecture

This section presents a brief overview of the Service Desk architecture and the role of Web API in that architecture.

As an interactive application, Service Desk uses a three-tier architecture:

- Client: communicates with the application using the ITSM Transaction Protocol (ITP) protocol. ITP is a communication protocol that is specific to Service Desk.

- Management Server: communicates with the database server using the Java Database Connectivity (JDBC) API.

- Database server: provides access to the database.

Logically, Service Desk is functionality divided over a number of layers, as shown in Figure 1-1.

Figure 1-1   **Service Desk Logical Layers**
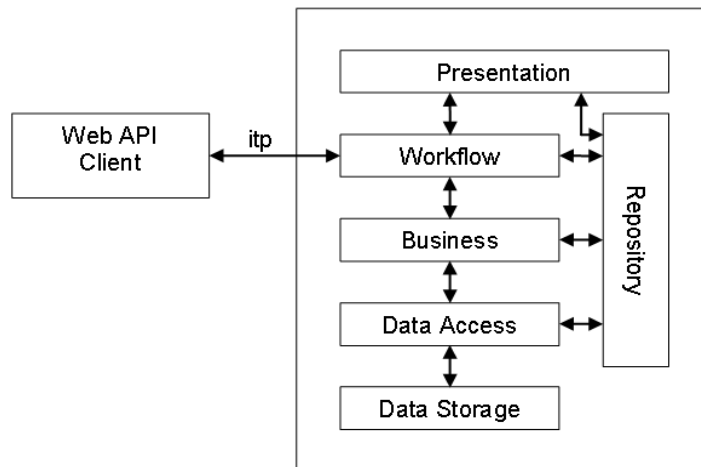
The presentation layer displays information and allows users to enter information. From an application point of view, the presentation layer embodies very little logic. All Service Desk-specific concepts are implemented in the workflow layer.

The workflow layer is the implementation of the Service Desk object model, and of application logic that is expressed in terms of the object model. The base classes of the workflow model implement the most frequent types of relation between sets of objects. Some of these relations can be found in almost any business application, such as one-to-one and one-to-many relationships. Other relations, such as history lines, are more specific to Service Desk and workflow applications.

The workflow layer maintains the state of the Service Desk application for a particular user that logs into Service Desk. All information that passes from users to the database passes through the workflow layer. This allows the workflow layer to enforce the logic of the business model.

The workflow layer also retrieves data for users. For example, it retrieves objects referenced by objects retrieved by the presentation layer from a user request.

The business and data access layers implement the interaction of the workflow layer with external resources, such as the database and mail delivery agents. The business layer in the Management Server, as defined in Service Desk, accesses the resources from the workflow layer. It also executes database rules.

In the Service Desk application, the presentation and workflow layers are physically situated in the interactive client application process. Because both the presentation and workflow layers use the repository, the interactive client consults the repository over the network the first time that it needs repository information. Repository information is cached, so the overhead of retrieving repository information over a network connection is minimized. The business and data access layers are executed in the Management Server. Most of the user log-on session state information is kept in the graphical user interface (GUI) client process. The Management Server process uses a small amount of state information to monitor the log-on session and authorization.

From the Service Desk perspective, a Web API application is a client of the Service Desk Management Server, even when it serves a different client. However, the distribution of the logical layers over the physical processes is different because the workflow layer is executed in the Management Server. This means that the client application does not need to query the repository to implement the Service Desk object model.

The Web API offers a set of Java classes and methods that implement the concepts of the Service Desk object model by remotely activating the workflow layer in the Management Server.

As a result, although many of the functions in the Web API look like database functionality, the logic of the workflow layer is activated when you submit a whole object to be saved using the Web API. The object is treated as if the values come from a form in the Service Desk client GUI application.

The business layer executes database rules in the same way as with the GUI client. However, UI rules are not relevant when you use the Web API because interactive per field data validation is not implemented by the Web API. You must write custom code in your Web API application to validate individual field values.

The Service Desk agent gets its commands from the Management Server. Even when the applications it executes are interactive, it is not relevant to the Web API architecture. In other words, an application started by the agent that uses the Web API has the same features and restrictions that apply to any other application. However, there are some unique problems with logging. For details, see "Be Careful with Logging" on page 63.

Installation

For convenience, the distribution contains Sun JavaDoc documentation, and a set of examples that illustrate the use of the Web API. It is recommended that you use them as a starting point for developing your own Web API applications.

The Web API is delivered as a single Java `.jar` file.

The procedure to run the installation program, extract the examples, and check the Java class path are described below.

Requirements

To build and test your own extensions to Service Desk you need a working Service Desk installation. This does not need to be on the same machine that runs the Web API applications. To compile and run Web API applications, you need the Sun Java 2 Development Environment, a working Web API installation on the client machine, and the Apache Ant Java-based build tool. For more information about the Apache Ant Java-based build tool, see the Apache Ant web site: http://ant.apache.org/. The examples assume that the demo data is present in the database.

Generating the Web API

If you are using a non-standard object model, you can use the Web API builder to build your own Web API. To do so, run the `OvObsWebAPIGenerator.bat` file in the installation `bin` directory. Shut down local running Service Desk sessions to save resources. The files generated by this step are the Web API source files, stored in the following directory:

`$INSTALL-DIR\data\web-api-generated\`

To Build the Web API jar file, you must compile the sources to a jar file. The following jar files are needed to do this:

- `OvObsWebApi-Client.jar`
- `OvObsWebApi-Common.jar`
- `OvObsSDK.jar`

- `OvObsWebApi-Server.jar`

Make sure that these files are available in the `$installdir\java` directory and are on the CLASSPATH. A sample compilation script `build.xml` is provided in the following directory:

`$Installdir\examples\webapi\generation`

To compile the sources to the web API jar file, copy `build.xml` to the following directory and run the "Ant" build tool:

`c:\Program Files\HP OpenView\data\web-api-generated`

This outputs the `ov-webapi-gen.jar` to the same directory.

Running the Installation Program

The Web API installation location for Microsoft Windows is `C:\Program Files\HP Openview\examples\webapi`. On UNIX, this is `/opt/OV/examples/webapi`.

Microsoft Windows
Locate and run the `setup.exe` installation executable in the Web API distribution.

Sun Solaris
Locate and run the `setup.bin` file in the Web API distribution.

HP-UX
Locate and run the `setup.bin` file in the Web API distribution.

Linux (using rpm)
Locate and run the `setup.bin` file in the Web API distribution.

Extracting the Examples

Select a location for the Web API examples. Substitute your chosen location in the extraction instructions described below.

Microsoft Windows
Double-click the `web-api-examples.zip` file, located in `C:\Program Files\HP Openview\examples\webapi`.

Alternatively, use the following command:

```
jar xvf Program Files\HP
Openview\examples\webapi\web-api-examples.zip
```

UNIX
Run the following command:

```
cd && mkdir -p web-api && cd web-api && tar xvf
/opt/OV/examples/webapi/web-api-examples.tar
```

Checking the Java Class Path

In a standard Sun Java Development Kit (JDK) environment, you can add the full name of the `.jar` file to the CLASSPATH environment variable, or you can explicitly add it to the class path in the invocations of the compiler and the applications. Refer to your Java Development Environment documentation for information on how to add the Web API `.jar` file to its class path. To use the Web API `.jar` file with the Sun JDK, you can use commands similar to the following:

On Microsoft Windows:

```
set CP="C:\Program Files\HP Openview\Java\sd-webapi.jar"
javac -classpath %CP% com\hp\ov\sd\webapi\examples\Example1
java -cp "%CP%" localhost jeffp servicedesk
```

On UNIX (Syntax for Bourne Shell at ksh or bash)

```
CP="/opt/ov/java/sd-webapi.jar"; export CP
javac -classpath $CP com\hp\ov\sd\webapi\examples\Example1
java -cp $CP localhost jeffp servicedesk
```

Compiling the Examples

Open the `compileExamples.bat` (Microsoft Windows) or `compileExamples.sh` (UNIX) file, and check that the class path in the commands points to the `.jar` file in the `\HP OpenView\Java` folder.

Run the compile script to compile the examples. This checks that the installation was successful. All programming examples should compile without problems. If you experience problems, check that:

- JDK and Sun Java Runtime Environment (JRE) are installed.

- The programs in the JDK `bin` directory can be found using the PATH environment variable.

- The value of the JAVA_HOME environment variable points to the JDK.

- The JDK and JRE versions match.

Running the Examples

Open the `runExample1.bat` (Microsoft Windows) or `runExample1.sh` (UNIX) file, and check that the class path in the Java command points to the `.jar` file in the `\HP OpenView\Java` folder.

Run the script. Check that the errors you receive are from the application and not the Java environment.

For Java or connection related error messages:

- Check the issues relating to the compiler described above.

- Check the class path in the script.

- Check the server name in the script.

- Check that the Service Desk application is running on the server machine.

For applications or permission related error messages:

- Check that the Service Desk application is running on the server machine.

- Check that the demo data was installed in the Management Server database.

- Check that you have a valid Service Desk account for the Management Server. It does not need to be a UI account. Not using the system account forces you to consider entitlement from the beginning. The scripts that invoke the example programs use accounts and data from the Service Desk demo database.

You have now installed a simple Web API development environment. Although you can copy its contents to your preferred Java development environment, the guide now assumes that you are using the Sun JDK.

NOTE   The compile script, `compileExamples.bat` (Microsoft Windows) or `compileExamples.sh` (UNIX), compiles the example programs only. For development purposes you need to build files that can be used with `make` or `ant`. You can also add a `-g` switch to the compiler command to include information in the class files for a Java debugger, such as jdb.

JavaDoc Documentation Tree

A JavaDoc document for the Service Desk Web API is provided on the Service Desk distribution CD in the `Doc` folder. The `Web API Javadoc.zip` file contains the set of hyperlinked HTML files that are generated from the API source code. These files describe the classes, interfaces, and methods of the API.

You can extract the files in the JavaDoc zip file to any location. Select the "Use folder names" option for the extraction. To open the JavaDoc, open the `index.html` file in the `html` subfolder (created in the folder you specified as the target location).

Your First Web API Application

This section presents a simple Web API application. It is a simplified version of Example 2. This application logs into the Service Desk Management Server, retrieves a service call by number, and displays some properties of the service call. The source for this example is part of the distribution.

The annotations for this code are shown in Table 1-1.

```
package com.hp.ov.sd.webapi.MyFirstWebApiApplication;


//1

import com.hp.ov.webapi.Session;

import com.hp.ov.webapi.sd.IServicecall;

import com.hp.ov.webapi.sd.IServicecallHome;


/**

 * MyFirstWebApiApplication

 *

 * A very simple web-api client application for demonstration purposes.

 *

 * Opens a session to the ServiceDesk and retrieves the description of

 * a service call with the functional ID of 1234567.

 */

public class MyFirstWebApiApplication {

    public static void main(String args[]) {

        Session session;
```

```
//2     try {
            session = Session.openSession("localhost", "olsek", "servicedesk");
        } catch (RuntimeException e) {

//3         System.out.println(e.getMessage());
            return;
        }

//4     IServicecallHome servicecallHome = session.getServicecallHome();

        IServicecall serviceCall;

//5     try {
            serviceCall = servicecallHome.openServicecall(1234567);
        } catch (RuntimeException e) {
            System.out.println(e.getMessage());
            return;
        }
//6     String labelDescription = servicecallHome.getLabelDescription();
//7     System.out.println(labelDescription + ": " +
        (serviceCall.getDescription()));
    }
}
```

Table 1-1　　　　**Java Source Code Annotations**

No.	Explanation
1	To use the Web API, you must import the classes with the basic types and interfaces of the Web API.

Table 1-1   **Java Source Code Annotations (Continued)**

No.	Explanation
2	Open a session on the Management Server on the current machine. User "olsek" with the password "servicedesk" is provided in the Service Desk demo database.
3	If logging in fails, the Web API or the Management Server throws an exception. It does not return null. The Web API only throws the RuntimeException. The message in the exception should be enough to present to users. In all cases, the level of severity is high enough to try to exit elegantly from the application.
4	Get the home for service calls from the session.
5	Retrieve a service call (with ID 1234567). Retrieve the object from the database and catch possible run-time exception. In this example: print a message and exit.
6	Query the "service call home" for a label to identify the description field of the service call. It will retrieve the appropriate label from the appropriate language pack.
7	Print the description field of the service call.

The first step is to log in to Service Desk. You must provide a network location where Service Desk can be reached, a user name and password. After successfully logging in, the Web API returns a session instance that implements a connection to the workflow layer in the Management Server.

When a session is established you can use it to query the session for the service call home object. For details about home objects, see "Programming Considerations" on page 60. For this example, treat the entity home as an interface to the database that allows you to store and retrieve objects from the database.

Web API Examples

The Web API comes with a number of example programs that illustrate
its use. The code is explained in Java comments in the source files.

To compile `MyFirstWebApiApplication` run
`compileMyFirstWebApiApplication.bat` (Microsoft Windows) or
`compileMyFirstWebApiApplication.sh` (UNIX).

To run the application run `runMyFirstWebApiApplication.bat` or the
`runMyFirstWebApiApplication.sh` (UNIX).

To compile the examples run the `compileExamples.bat` (Microsoft
Windows) or the `compileExamples.sh` (UNIX) file. Use the appropriate
batch or script file to run the example. For example, to run
`example1.java`, use `runExample1.bat` (or `runExample1.sh`). The
examples are explained following part of this section.

Additional application-specific examples are provided in the
`com.hp.ov.sd.webapi.cases` package. After installing the example
tree, the application case source and class files are stored in the folder
where you unpacked the files, for example the `c:\webapi`. These
examples are described at the end of this section. Use the
`compileCases.bat` file (or `compileCases.sh`) to compile these examples.
Use the appropriate batch or script file to run the example case. For
example, to run `RelateSCtoChange.java`, use `runCase1.bat` (or
`runCase1.sh`). For every Microsoft Windows batch file in the following
table, a UNIX shell script file is also provided.

Table 1-2   **Web API Example Files**

File	Description
`compileMyFirstWebApiApplication.bat`	Compile the MyFirstWebApiApplication example
`runMyFirstWebApiApplication.bat`	Run the MyFirstWebApiApplication example
`compileExamples.bat`	Compile all examples; `example1.java` - `example9.java`

Table 1-**2**   **Web API Example Files (Continued)**

File	Description
`runExample1.bat`	Runs example 1. (Run `runExamplex.bat` to run example x.)
`compileCases.bat`	Compiles all use case examples.
`runCase1.bat`	Runs the RelateSCtoChange use case.
`runCase2.bat`	Runs the RelateSCtoProblem use case.
`runCase3.bat`	Runs the SetFolderToCaller use case.
`runCase4.bat`	Runs the SetSLA use case.

Example1.java

How to open a session to the Service Desk Management Server. Retrieves the account information object from the session object for the user that owns it. Extracts and prints properties of the account. Lists all persons that use this account, illustrating how to obtain a set of related objects using a one-to-many relationship.

Example2.java

How to open an existing service call, based on its functional ID. Retrieves the history lines belonging to the service call. One of the properties of a history line is the date of its creation. The Web API offers methods to represent Java dates as strings. The example illustrates one of these methods.

Example3.java

How to create a new problem, set some properties of the problem, and save it. Optionally, you can provide the name of a template. If the specified template exists and is a template for problems, the template is used to initialize the new problem. Finally, a history line is added to the problem.

Example4.java

How to open an existing incident, change the information field of the incident, and assign the incident to a person.

The example illustrates application-defined queries built with a `where` clause and search criterion objects. It shows how to access the aggregated object in the incident and how to update the assignment. Aggregated objects are stored in the same database record as the object to which they belong. Finally, the incident is saved to the database, illustrating how modified objects, including the aggregated subobjects, are saved.

Example5.java

Handcrafted selections with `where` clauses and criteria. The selection is for a date field and illustrates some date manipulation methods.

Example6.java

Illustrates the concept of a view. Views are a means to reduce the amount of information in a home object that is visible to the application. Unlike selections that are made with `where` objects and search criteria, views are not built dynamically.

The description of a view is stored in the internal system. The application refers to a view in the internal system to retrieve the information that belongs to the view. In the Service Desk application, views can limit the number of available fields. In this example, only the selection of the set of objects that is retrieved through the view is relevant. Using a view enables you to define complex filters using the Service Desk GUI, while the programming for selecting the data using the API remains simple.

Example7.java

How to use web application profiles to control the set of attributes returned by the methods that return an array of entity instances. It illustrates the concept of the IWebApiApplication, and demonstrates its importance for performance.

Example8.java

Illustrates the use of many-to-many relationships in the Service Desk Web API. Scans the membership relationship between persons and workgroups. Creates a new workgroup and relates some persons to it. Finally, it deletes the new workgroup.

Example9.java

How to use reflection using the `IEntityInfo` interface to get a display name for an entity type, and to get a list of applicable templates.

RelateSCtoChange.java

Relates an existing service call in the database to an existing change. Use `runCase1.bat` (or `runCase1.sh`) to run this application case with the Service Desk demo database.

RelateSCtoProblem.java

Relates an existing service call in the database to an existing problem. Use `runCase2.bat` (or `runCase2.sh`) to run this application case with the Service Desk demo database.

SetFolderToCaller.java

Searches for the caller of an existing service call and then sets the folder of the service call to that of the caller or the caller's organization. Use `runCase3.bat` (or `runCase3.sh`) to run this application case with the Service Desk demo database.

SetSLA.java

Searches for the caller of an existing service call and then sets the service level agreement of the service call to that of the caller or the caller's organization. Use `runCase4.bat` (or `runCase4.sh`) to run this application case with the Service Desk demo database.

2 Programming Interface

This chapter describes in detail the structure of the HP OpenView Service Desk Web API. It presents the classes and methods that are available to access and manipulate HP OpenView Service Desk data. It also explains exception handling and the use of views.

Web API Structure

The Web API consists of two parts: the server-side component and the client-side component. The server-side component runs on the Service Desk management server, in the same Java Virtual Machine (JVM) process as the Service Desk server application. Its task is to handle incoming calls from the client-side component.

The client-side component runs in a separate Java virtual machine, in a pure Java environment. This could be the same process where a web server is running. The client-side component consists of the application programming interface (API) classes and it is this component that is explained in this document.

There are three levels of client-side classes:

- Session
- Home
- Entity

The first level consists of the session classes. You use the `Session` class to start up a session and access the functionality of the API.

The second level consists of the home classes. Each entity type has its own home class, `I<Entity Name>Home`. For example, `IChangeHome` or `IServicecallHome`.

You can use these classes to do the following:

- Open individual records of the entity to which the home class belongs
- Create new records
- Find records using search criteria

The home classes use two kinds of utility classes, called `I<Entity Name>Where` and `I<Entity Name>Entitlement`. Use the `I<Entity Name>Where` class to build `where` clauses for a find action and the `I<Entity Name>Entitlement` class to provide information about entitlement.

The third level consists of the entity classes. Each entity type has its own entity class I*<Entity Name>*. **For example,** IChange **or** IServicecall. **These classes contain the values of individual record attributes, as well as the methods to manipulate these values.**

Programming Concepts

This section does not attempt to describe the Service Desk object model. Probably the best way to gain an understanding of this is to browse the Service Desk client tool. The links from one piece of information in the application to another correspond to the relationships in the object model.

Service Desk Object Model

A detailed and navigable description of the Service Desk object model can be found on the Service Desk distribution CD in the following file:

`doc\Data Dictionary\Data_Dictionary_Items.htm`

This file contains extensive information. Because it is generated from the same material as the Web API, it is bound to match the Web API. To locate specific information about the Data Dictionary, it is often easier to use Sun JavaDoc documentation for the Web API.

Interfaces

The classes of the Service Desk object model appear as interfaces in the Web API. With an few exceptions, you can use the instances of the interfaces as if they were instances of the corresponding class. The Web API is implemented with interfaces instead of classes to force you to use the homes as an instance factory.

Entities

The Service Desk programming concepts are based on the concept of an entity. For all practical purposes, `Entity` is the base class of the Service Desk object model. All objects that are saved to the database are instances of the `Entity` class. Depending on the context, instances of subclasses of `Entity` are referred to as entities or `Entity` instances. The `Entity` class (`IApiEntity` in the Web API), together with the corresponding `EntityHome` class, saves and retrieves instances in the database.

Entity Home

The entity home is a fundamental Service Desk concept. The instances of the instantiable classes in the Service Desk object model are saved in the database tables. The classes of objects that are responsible for saving entities to the database are called entity homes.

Additional tasks of entity homes include retrieving entity instances from the database, and creating fresh entity instances that still have to be saved to the database. Therefore, the entity home is at once a factory of entity instances, and an interface to the database to store and retrieve the instances.

Naming Conventions

Except for logon and logoff, the Web API software is generated from the object, and follows a rigid naming convention. With this naming convention, it is relatively easy to infer the name of a method or a class from the Data Dictionary.

This section describes the most important families of classes and methods. It does not describe every class and method. The JavaDoc provided with the WebAPI contains extensive information about classes and methods. It is not the intention to describe every class and method. The JavaDoc provided with the Web API contains extensive information.

The structure of the Web API follows certain paradigms. The existence, name and functionality of a method are applications of the paradigms. The descriptions that follow deal with the families of classes and methods that come from a certain paradigm. Some of the properties of the Service Desk classes are discussed in general, together with the functionality of the methods.

Most of the class, attribute, and method names used in the following discussion are not actual names of classes. Rather, they refer to the patterns that the paradigms follow. In particular, the words `Entity`, `Attribute`, `SimpleType`, `RelatedEntity`, `ReffedEntity`, `ContainedEntity`, `StringAttribute` (and some grammatical inflections) do not refer to actual concepts. You should replace these words with the name of actual entities, attributes, and types. For example, for every entity `Entity`, the name of the interface that the class implements is `IEntity`, and the name of the interface to the entity home is `IEntityHome`. Names that refer to this convention are used

throughout the rest of this section. Usually, this notation is used as if the names refer to actual interfaces or methods, without mentioning that the name actually refers to a naming convention rule.

Entity Interface

As previously explained, entities are the equivalent in the interface to the classes in the Service Desk object model. The JavaDoc documentation that accompanies this guide describes a large number of entity interfaces.

The most important families of entity classes are shown in Table 2-1.

Table 2-1   **Entity Class Families**

Family	Description
IWorkflow	Note that all objects, not just Workflow instances, are managed by the workflow layer. Also: • They have a functional record ID (for example, Service Call 1024). • They contain aggregate entities for assignment and history. • No system constraints on deletion exist. It is transient data, although user access and status restrictions can be implemented.
Non-workflow	• Mostly identified by a search code rather than a functional ID (CIs have both). • System constraints on deletion, static data defining the base data in the Service Desk implementation at the customer's site. For example, managed configuration items (CIs), services, Person and Organization data, and so on.

Table 2-1   **Entity Class Families (Continued)**

Family	Description
Codes	• For example, status and category texts. • A list of codes can be searched from the home class you get from the entity home. For example, you can get a list of statuses from `IStatusProblemHome` that you can use for an `IProblem` instance.
Aggregated entities	Can be accessed only using another entity. Many types of aggregated entities can be found inside different kinds of containing entities. For example, `IAssignment` instances are aggregated inside `IProblem` instances and inside `IServicecall` instances.

Attributes

Entity instances have attributes. Each attribute has its own `getAttribute`() and `setAttribute`() method. The `getAttribute`() method returns the value of the attribute, and the `setAttribute`() method allows you to give the attribute a value. The Java type of the attribute values can be derived from the method definitions.

Attributes can be classified into the following groups:

• Attributes that have a basic Java type. For example, String, Long, Boolean, and so on. The entity classes have `getAttribute`() and `setAttribute`() methods for these attributes.

• Attributes that are references to other entities. These attributes are called entity reference attributes. The entity classes have `getReferredEntity`() and `setReferredEntity`() methods for these attributes.

• Aggregated entities that show up as an attribute. A reference to the aggregated entity can be obtained with the `getContainedEntity`() methods list in Table 2-2. Changes to the contained entity can be transferred back to the parent instance with the `transfer`() method of the aggregated entity instance.

- Attributes that reference a set of entities. For example, history lines. These attributes are called entity set reference attributes. To retrieve an array with references, use the getRelatedEntity() methods listed in Table 2-2. To sever a relationship, use the unrelateRelatedEntity() methods. The related entity instances have a getParentEntity() method to follow the reference in the opposite direction.

- Many-to-many (n-m) relation attributes. These are actually a special kind of entity set references.To retrieve an array with references, use the getRelatedEntity() methods in Table 2-2. To relate one object to another, use the addRelatedEntity() methods. To sever a relationship, use the unrelateRelatedEntity() methods. To navigate the relationship in the opposite direction, the related entities offer the same kind of methods.

Methods

This section provides an overview of the methods defined for the entities. Not all methods exist for all entity types.

IEntity Interface Methods

The IEntity interface methods are shown in Table 2-2.

Table 2-2　　　　**The IEntity Interface Methods**

Attribute	Description
`void`	`addRelatedEntity(`IRelatedEntity object`)`
	(Where `Entity` has a one-to-many relationship to `RelatedEntity`)
	Adds the `RelatedEntity` object to the set of `RelatedEntity` objects that belong to this `Entity` instance. The `Entity` instance must exist in the database. The `RelatedEntity` object is saved to the database by this method. Methods of this kind exist for all one-to-many relationships.
	Because this is a database action, it can fail. Catch RuntimeExceptions when you invoke this method. In addition to physical database I/O failure, there are many reasons that a database modification can fail. Consider validating the attribute values, referential integrity, entitlement, and business logic.
	The related entity is added to the database immediately. In the case of history lines that describe changes to the entity itself, this means that the update of the entity itself, and the insertion of the history line, are not executed as a single database transaction.

Table 2-2   **The IEntity Interface Methods (Continued)**

Attribute	Description
`void`	`addRelatedEntity(`IRelatedEntity object`)` (Where `Entity` has a many-to-many relationship to `RelatedEntity`) Adds the `RelatedEntity` object to the set of `RelatedEntity` ojects that are related to this `Entity` instance. Methods of this kind exist for all many-to-many relationships without any attributes. Relationships with attributes are implemented as an entity type. The relationship is saved to the database immediately. The current entity instance and the related entity instance should be saved to the database before they are related. This is a database action, so catch exceptions.
`void`	`unrelateRelatedEntity(`IRelatedEntity object`)` Remove `RelatedEntity` object to the set of `RelatedEntity` objects that are related to this `Entity` instance. Methods of this kind exist for all many-to-many relationships without any attributes. Relationships with attributes are implemented as an entity type. The relationship is removed from the database immediately. The current entity instance and the related entity instance should be related to allow you to unrelate them. This is a database action, so catch exceptions.
`SimpleType`	`getAttribute()` Gets the value of `Attribute`. `getAttribute()` methods exist for all attributes that have a simple type. For example, the type of the attribute is not an entity type from the Service Desk object model. If the attribute is not set, `getAttribute()` returns null. (No exceptions are thrown.)

Table 2-2   **The IEntity Interface Methods (Continued)**

Attribute	Description
`IRelatedEntity[]`	`getRelatedEntity()` **Loads all** `RelatedEntity` **objects that belong to the** `Entity` **instance from the database into an array. Methods of this kind exist for all one-to-many relationships. Because this is a database action, it can fail. Catch RuntimeExceptions when you invoke this method.**
`IReffedEntity`	`getReferredEntity()` **Gets the entity instance to which this attribute refers. This attribute follows the link in the database to a different entity instance. These methods exist for all something-to-one relationships in the object model.** `getReferredEntity()` **returns null if no referred entity exists. Because following the link (if it exists) involves database I/O, catch exceptions when you invoke methods from this family.**
`IContainedEntity`	`getContainedEntity()` **Gets the aggregated entity instance that is part of the current entity instance. The contained entity instance is stored in the same database record as the containing entity instance. If you make changes to the return value that you want to change, use its** `transfer()` **method to transfer them back before you** `save()` **this instance.**
`void`	`setAttribute(`SimpleType value`)` **Sets the value of an attribute.** `setAttribute()` **methods exist for all attributes that have a simple type. The type of the attribute is not an entity type from the Service Desk object model.**

Table 2-2   **The IEntity Interface Methods (Continued)**

Attribute	Description
void	setReferredEntity(IReferredEntity value) **Makes the current instance refer to a different entity. You must invoke a** save() **method to save the new reference to the database. Also, no** setContainedEntity() **methods exist. An aggregated entity can be changed, and its changes can be saved to the database through** transfer() **methods, and a subsequent** save() **method. Because the aggregated instance does not exist independently from its container, it is logically impossible to just change the reference.**
void	cancel() **Cancels previous programmatic changes. The properties of the entity instance are reset to their original values.** To cancel the changes, cancel() **must be called before the changes are committed to the database with** save(). **The Web API does not use the database to reset the values, so it is not absolutely essential to catch exceptions when you invoke this method.**
void	delete() **Delete the entity instance from the database. Because this is a database action, it can fail. Catch RuntimeExceptions when you invoke this method. In addition to physical database I/O failure, there are many reasons that a database modification can fail. Consider validating the referential integrity, entitlement, and specific checks in the business logic. For obvious reasons, there is no** delete() **for aggregated entities.**

Table 2-2   **The IEntity Interface Methods (Continued)**

Attribute	Description
com.hp.ov.obs.OID	getOID()
	Gets the unique object ID value that identifies this entity instance. No two instances from the database return the same value for getOID(). The value returned by getOID() is persistent. That is, it remains the same over time for the same instance. It can even be used to retrieve a particular instance from the database. OIDs are internal to the application. They are not intended for storage or display by a Service Desk application. One of their few sensible uses in application code is to determine whether two entity instances are the same instance in the database.
	OIDs is also used to refer to instances from a list of values. Do not confuse object IDs with functional IDs that do make more sense in application code.
Java.lang.Long	getID()
	Gets the functional ID of the object. Similar to the object ID, the functional ID uniquely identifies the object. Only entity types that inherit from Workflow have functional IDs. Functional IDs are intended to uniquely identify the instances to users. They are an immutable attribute. In contrast, the object OIDs are used by Service Desk itself to identify entity instances.

Table 2-2   **The IEntity Interface Methods (Continued)**

Attribute	Description
`java.lang.Boolean`	`isDeleteAllowed()` Returns true if you are allowed to delete this entity instance. The return value depends on the user that is logged in, and the entitlement-related properties of the entity instance. This method is used to avoid the `delete()` method throwing an exception. In addition to entitlement, there are many reasons that deletion can be forbidden. These situations are not checked by the `isDeleteAllowed()` method, so you have to catch exceptions in the `delete()` invocation.
`java.lang.Boolean`	`isModifyAllowed()` Returns true if you are allowed to update this entity instance. The return value depends on the user that is logged in, and the entitlement-related properties of the entity instance. This method is used to avoid the `save()` method throwing an exception. In addition to entitlement, there are many reasons that saving can be forbidden. These situations are not checked by the `isModifyAllowed()` method, so you have to catch exceptions in the `save()` invocation.

Table 2-2   **The IEntity Interface Methods (Continued)**

Attribute	Description
`void`	`save`() Saves the changes that you have made to this entity instance. Because this is a database action, it can fail. Catch RuntimeExceptions when you invoke this method. In addition to physical I/O failure, there are many reasons that database modification can fail. Consider validating the attribute values, referential integrity, entitlement, and business logic. Aggregated entities do not have a save() method. Use their transfer() method as well as their parent's save() method.
`void`	`transfer`() Propagates changes that are made to the entity to which this aggregated entity belongs. Use the `save`() method of the parent entity to save the changes in the database.
`IReferringEntity[]`	`getReferringEntity_ReferringAttribute`() Retrieves all `ReferringEntity` instances that refer to the current entity instance by using their `ReferringAttribute` attribute. This attribute is a shortcut for `IReferringEntityHome.searchOnReferringAttribute( this )`.

IEntityHome Interface Methods

Home interfaces give descriptive information about the entity type. For many entities, the home is the interface to the database that stores them. The home can also be the factory object that can be used to make new entities. The IEntityHome interface methods are shown in Table 2-3.

Table 2-3   **The IEntityHome Interface Methods**

Attribute	Description
void	delete(com.hp.ov.obs.OID id) **Deletes the entity instance with the specified ID. Because this is a database action, it can fail. Catch RuntimeExceptions when you invoke this method.**
com.hp.ov.obs.OID	getDefaultView() **Returns the default view for this particular type of entity. If there is a default view, it can be used to load an array of instances using the** findEntities() **method.**
com.hp.ov.obs.OID	getEntityID() **Returns a long value that identifies the entity type that is stored uniquely using this** EntityHome.
boolean	isAggregated() **Returns true when instances of this entity type are always part of the instance of another entity type. An aggregated entity instance cannot exist independent of another entity instance. This means that the** EntityHome**s of aggregated entities do not support the database-related functionality or the methods to make new entity instances.**

Table 2-3   **The IEntityHome Interface Methods (Continued)**

Attribute	Description
IEntityWhere	createEntityWhere()
	Creates a where object to retrieve Entities from the database. The IEntityWhere object is used to contain search criteria that define a selection. It also loads an array of entity instances that match the criteria with the findEntity() method.
IEntity[]	findAllEntities()
	Loads all Entity instances in the database into an array. Do not use any selection criterion. For many types of entities, the number of instances may be considerable.
IEntity[]	findEntity(IEntityWhere where)
	Loads selected Entity instances from the database into an array. Use the selection criteria from a "where" clause selection.
IEntity[]	findEntity(com.hp.ov.obs.OID view)
	Loads selected Entity instances from the database into an array. Use the selection criteria from a view.
IEntityEntitlement	getEntityEntitlement()
	Gets entitlement for this Entity type. Entitlement objects can be used to determine whether the current user is allowed to change attributes of entity instances, and to determine which attributes are required attributes. For details, refer to the IEntityEntitlement interface.

Table 2-3   **The IEntityHome Interface Methods (Continued)**

Attribute	Description
`java.util.Hashtable`	`getEntityViews`() Gets all views of `Entity`. Views are predefined selections to retrieve a set of instances. Views are defined in the Service Desk administrator's console.
`java.lang.String`	`getLabelAttribute`() Gets the label that can be used with the value of the Attribute. A `getLabelAttribute`() method exists for all attributes of the entity type. When a label must be shown on a form, this method can be used to retrieve the label name.
`IEntity`	`openEntity`(`com.hp.ov.obs.OID oid`) Retrieves an existing entity instance from the database. Because this is a database action, it can fail. Therefore, catch the possible RuntimeException. If no `Entity` with this particular object ID exists, a RuntimeException is thrown. OIDs are internal to Service Desk applications. It rarely makes sense to use OIDs in application code. A possible use of this method is to retrieve individual instances from a list of values. Do not confuse OIDs with functional IDs that do make more sense in an application context. CAUTION: The methods that retrieve an entity by functional ID and by object ID differ only in the type of the argument.

Table 2-3   **The IEntityHome Interface Methods (Continued)**

Attribute	Description
IEntity	openEntity(long id) Retrieves an existing entity instance from the database. Because this is a database action, it can fail. Therefore, catch the possible RuntimeException. If no Entity with this particular functional ID exists, a RuntimeException is thrown. Do not confuse functional IDs (that are specific to application data) and IWorkflow instances with object IDs (that are internal to the operation of Service Desk). CAUTION: The methods that retrieve an entity by functional ID and by object ID differ only in the type of the argument.
IEntity	openNewEntity() Returns a fresh instance of Entity. The entity instance is not saved to the database before its save() method is invoked.
IEntity	openNewEntity(com.hp.ov.obs.OID template) Returns a fresh instance of Entity. The fresh instance is initialized according to the template in the argument. The entity instance is not saved to the database before its save() method is invoked.

Table 2-3 **The IEntityHome Interface Methods (Continued)**

Attribute	Description
IEntity[]	searchOnAttribute(SimpleType value) Searches for Entity instances, where attribute Attribute has a specific value, and load them into an array. The type of the value is the type of the attribute. In many cases the searchOnAttribute() methods return more entity instances than you ever want to retrieve. Use them with discretion.
IEntity[]	searchOnAttribute(IReferredEntity value) Searches for Entity instances, where attribute Attribute refers to a specific instance of an entity, and load them into an array. The type of the value is the type of the attribute. In many cases the searchOnAttribute() methods return more entity instances than you ever want to retrieve. Use them with discretion.

IEntityWhere Interface

Instances of this interface are used to load selected Entity instances from the database. To make a selection from the database, you create an EntityWhere object. You then add search criteria on the attributes to the EntityWhere. Finally, you pass the EntityWhere instance to the findEntity() method of the EntityHome object to load an array of

Entity instances. The instances that are loaded match all the criteria that have been added to the EntityWhere. The methods of the IEntityWhere interface methods are shown in Table 2-4.

Table 2-4   **The IEntityWhere Interface Methods**

Attribute	Description
void	addContainCriteriumOnStringAttribute(java.lang.String value) Adds a contains-string-criterion on the value of a string attribute. The value of the string attribute of all instances that will be retrieved from the database must contain the string value.
void	addCriteriumOnContainedEntity(IContainedEntityWhere where) Restricts the selection to those instances where the aggregated instances match the selection on the contained entity type in the "where" argument.
void	addCriteriumOnReferredEntity(IReferredEntity value) Restricts the selection to those instances that refer to referred entity instance value.
void	addCriteriumOnAttribute(SimpleType value) Adds an equal-to search criterion on the simple type attribute. The value of the attribute of all instances that will be retrieved from the database must have this value. AddCriteriumOnAttribute() methods exist for all attributes that have a simple type. That is, the type of the attribute is not an entity type from the Service Desk object model.

Table 2-4   **The IEntityWhere Interface Methods (Continued)**

Attribute	Description
`void`	`addCriteriumOnAttributeRange(`SimpleType`[]` `values)`
	Adds a range search criterion on the simple type attribute. The value of the attribute of all instances that will be retrieved from the database must be between the first two values in the values array or equal to one of them. `addCriteriumOnAttributeRange`() methods exist for all attributes that have a simple type. That is, the type of the attribute is not an entity type from the Service Desk object model. `addCriteriumOnAttributeRange`() also allows you to set less than or equal to (<=) or greater than or equal to (>=) criteria. If you pass null as the upper or lower bound in the values array, a selection that is bounded only at the other side is added to the `EntityWhere`.

IEntityEntitlement Interface

For every `Entity` type, an `IEntityEntitlement` interface exists. You can use it to determine whether the individual properties of instances can be modified. The `IEntityEntitlement` interface can also tell you whether the individual properties are required properties. The methods available using the `IEntityEntitlement` interface are shown in Table 2-5.

Table 2-5   **The IEntityEntitlement Interface Methods**

Attribute	Description
`boolean`	`isModifyAllowedAttribute`()
	Checks whether attribute `Attribute` can be modified. The return value depends on the user who is logged on and on the entitlement-related properties of the attribute of this type of entity instance.
	This method can be used to prevent the `save`() method from throwing an exception. In addition to the entitlement of a particular attribute for all instances of a certain entity type, there are many reasons that a `save`() can be forbidden. These situations are not checked by the `isModifyAllowedAttribute`() method. The modification of a particular instance might be forbidden, although the attributes of instances in general are freely changeable. Always try to catch exceptions when you `save`() an entity instance.
`boolean`	`isRequiredAttribute`()
	Checks whether `Attribute` is a required attribute. If you do not set the attribute, and it is not set by the template that you used to make an instance of this type of entity, saving the instance will fail.

The Session Class

The `Session` is the implementation of a session to a running Service Desk management server. Once you have a connection, you can use it to communicate with the server. The connection is to an instance of the workflow layer in the server that keeps state information for your client

application. To log on to a management server, you provide a username and a password that belong to an account. You can use any account that is authorized to use the Web API.

For an economic use of licenses and conservative security, it might be wise to create a special user for the Web API as a whole, or for the separate applications that use it.

A Web API application receives the permissions of the user who is logged on to the current session. If you are using the Web API as an extension mechanism to Service Desk, this may sound too restrictive. For a detailed discussion of Service Desk entitlement, refer to the *Service Desk Administrator's Guide*.

Accounts that belong to the named license, or to the concurrent users license groups, consume one of the licenses that you have.

Table 2-6   **Session Interface Methods**

Attribute	Description
`IEntityHome`	`getEntityHome`() Gets the home interface for entity `Entity`. Home interfaces give descriptive information about the entity type. For many entities, the home is the interface to the database that stores them and the factory object that can be used to make new ones.

`static Session`	`openSession(` `java.lang.String server,`
	
	 `java.lang.String username`
	
	 `java.lang.String password`)
	
	 This is the way to obtain a connection to a management server. The arguments are as follows:
	
	 • *server*: The server to connect to. This string argument can have the following formats:
	
	 — *computer_name*
	
	 — *IP_address*
	
	 — *computer_name:port*
	
	 — *IP_address:port*
	
	 • *username*: The username of the account you want to use. It must be a known Service Desk user who is allowed to log on.
	
	 • *password*: The password of the account you want to use.
	
	 This method can fail. Always catch the RuntimeException that it might throw.

Table 2-6   **Session Interface Methods (Continued)**

Attribute	Description
void	setApplicationSettings(IWebApiApplication application) Lets IWebApplication **application determine which attributes of the different entity types instances are retrieved immediately, and which attributes are retrieved only when they are actually used. The mechanism only controls the methods that return arrays of entity instances (that is, the** findEntity() **methods of the** IEntityHomes). **Settings are usually done once, at startup. There is no need to make them available afterwards, so a corresponding** getApplicationSettings **method is not offered.**
void	closeConnection() **Closes the connection that you have to a workflow layer in the management server. Connections that are not used are closed after a certain amount of time. If you want to clean up the unwanted files that you leave behind, you can use this method to free up resources in the management server immediately. This method makes the connection worthless, so it does no harm to forget the reference to the object as well.**

Exception Handling

The Web API throws only one kind of exception: the Runtime Exception. The Web API has been designed in such a way that exceptions are rare in a well-designed application. Exceptions that do occur are a fundamental part of the application logic.

An exception is more than just part of the programming interface. The message contained in the exception is intended for users. Usually, it is sufficient to display the message, and either restart the action as a whole or just to exit.

Subtle exception handling logic is performed inside the Web API classes. The exceptions that are thrown concern severe problems or the application logic itself. Because the message in the exception is aimed at users, it is provided in their language as soon as the Web API identifies their language. That is, as soon as a user is logged on.

What you do when the Web API throws an exception in your application is largely an issue of personal preference. This decision depends on the action, not on the exception.

Views

Views are a means to limit both the set of entity instances and the set of attributes of the instances that are retrieved from the database. Additionally, a view can have an ordering on certain attributes, and it can be grouped using certain attributes. In many respects, a view is comparable to an SQL "select" statement. Views can be defined in the Service Desk administrator's console. The Web API uses only the ordering and the selection on the rows of the view. With the Web API, limitations on the set of columns that is retrieved come from application settings, not from the views. The `IEntityHome.findEntity(com.hp.ov.obs.OID viewId`) method allows you to use views that you have defined in the administrator's console from your Web API application. Using views saves you from having to explicitly code the selection, and it gives better performance because your calls to make the selection do not have to be passed to the server over the network.

An additional mechanism to improve performance exists. The Service Desk administrator's console allows you to control which attributes of an entity instance are initially returned to the Web API client by the methods that return an array of entity instances. Because attributes that have not been returned initially have to be fetched one by one over the network, web application profiles are a powerful performance-tuning tool.

The settings are organized in named groups. The groups are available to a Web API application as `IWebApiApplication` instances. By default, only the object ID and a few attributes are returned when a set of instances is retrieved. The `IWebApiApplication` settings allow you to return more attributes with the array. The settings eliminate the need to fetch attributes that are needed later on individually. That is, those attributes that have not yet been returned with the array are retrieved when you ask for them. On the one hand, limiting the number of attributes reduces the amount of data that is retrieved from the database, and that is returned over the network. This improves performance in terms of client memory and network traffic. On the other hand, when your application retrieves attribute values one by one, its performance is degraded by individual calls over the network connection.

Returning exactly the attributes that you want (and nothing more) is optimal. Experience suggests that fetching too many attributes is less harmful than fetching too few.

Programming Considerations

The Web API is designed for transaction-based applications. This is typical of an intranet application in which users fill out forms and browse data. All the application logic is implemented inside transactions that receive the user input and return some data. The data is intended for the user's immediate consumption. Also, in a low to medium volume situation, the Web API can be used to implement database rule actions. This section highlights some of the restrictions on the Web API when used in non-typical scenarios.

Not Intended for Field-by-Field Validation

It is possible to develop applications that do not fit the scenario described above. However, the performance of these applications may well be less than optimal. If you want to use the Web API for purposes other than those for which it was designed, you should be aware of some of the disadvantages before you design and develop your application.

The Service Desk Web API is designed for transaction-based applications. Because most of the information that the Service Desk graphical user interface (GUI) client uses for field-by-field validation is available through the Web API, you could write your own field validation code and incorporate it into your application. However, UI rules are not available.

Because all Web API calls are made over the network, for performance reasons you need to limit the number of mini-transactions that you make to perform field-by-field validation. Also, even if you check user input extensively, you need to consider proper handling for catch exceptions.

Implementing Service Desk UI Rules

Not only is trying to use the Web API to implement UI rules for input validation in the Service Desk GUI client application very difficult, it also has severe disadvantages.

Because a Java Virtual Machine (JVM) has to be started, and the application has to log on to the management server, every time one of these small checks is performed, performance suffers.

In addition, because you access the same database as the GUI client, but using a different route, it will be very difficult to avoid database concurrency issues (such as locking conflicts) between the two programs, as well as failed transactions, because one program changes a database record that the other also owns.

Not Optimal for Bulk Exchange

Applications that use the Web API to update or consult the database on a much larger scale than online transaction processing may suffer from performance problems. The Web API allows you to add entity instances only to the database one by one, using the `IEntityHome.openNewEntity()`..`IEntity.save()` path. Feeding a large number of entities to Service Desk with the Web API causes Service Desk to validate every entity instance individually, and to perform the insertion of the individual entities as separate database transactions. For medium-volume data feed applications, this may not be a problem. However, when you use the Web API to feed a lot of data to Service Desk at once, performance suffers.

A similar restriction also applies in the opposite direction. The Web API returns sets of entity instances in arrays. For a high-volume-at-once application, the size of the arrays would become prohibitive. Either many attribute values are returned with the arrays, and the arrays consume a lot of memory, or the attributes are fetched over the network one by one. Neither situation is suitable for large-scale data extraction.

From the point of view of performance, Service Desk Data Exchange is superior to the Web API for bulk data exchange.

Authentication

Web API applications need to log into a management server with an account and password. Although you can use any account that is authorized to use the Web API, for an efficient uses of licenses and conservative security, it may be wise to create a special user for the Web API as a whole, or for the separate applications that use it.

A Web API application receives the permissions of the user who is logged on to the current session. Although these permissions might appear too restrictive when using the Web API as an extension mechanism to Service Desk, it is a restriction you need to be aware of when designing your Web API applications.

Web API Sessions are Not Thread-Safe

The state information and the ITSM Transaction Protocol (ITP) communication protocol that communicates with the workflow layer on the other side are not thread-safe. Therefore, do not share sessions between threads.

Actually, when you consider this restriction from a design perspective, you realize that it is almost unavoidable. When a session is used to communicate with the workflow layer in the management server, it builds up a context with a state. Another thread that uses the same session will change and probably destroy the state. Therefore, do not share sessions between threads. To avoid concurrency issues, do not use more than one session for a transaction.

Allow for Concurrency Issues

Service Desk uses optimistic concurrency control. This means that your application can read data that has been read by other users. The advantage of this is all data is accessible at every moment. The disadvantage is that Service Desk and the database system have to make decisions about the validity of the database updates at the moment that they are submitted. An object stored in the database that has been changed by another user before you have the possibility to save it is a possible cause for transactions to fail. Your application design must retry important transactions that fail.

Initialization is Expensive

A Java program that opens a Web API session, does a few simple things and then exits, has the advantage that it is easy to understand and easy to debug. However, if your application invokes this kind of program with varying frequency, there may be a performance problem. Loading and starting a Java Virtual Machine (JVM) is probably more work than executing your small program.

In addition, once your small program is started, logging on to the Service Desk management server is probably more expensive than the actual application actions of your small program. An application that frequently uses the Web API for relatively small transactions must not use separate Java programs that open and close sessions to the management server. You should try to build programs that exist longer and that use just one session during their whole use.

Be Careful with Logging

Some of the provided examples use System.out.println() to inform you of the execution progress. For simple command-line applications, this is the simplest and most appropriate way to produce output.

However, if you use the Web API to implement database rule actions or to write servlets, severe disadvantages exist. In the best case, your output can be lost. In the worst case, uncontrolled output will hang your server (for example, when the program pops up a dialog box and only continues after the "OK" button is pressed). Rule actions, programs that are started by the Service Desk agent, do not have their standard output connected to a suitable medium. Therefore, you should not use standard output.

If your application produces informative output of the System.out.println() type, you should make sure that it is redirected to a log file every time it is used as a rule action. There are concurrency issues with the log file. When two rule actions execute simultaneously, their output is mixed. To avoid this, use a unique file name. For example, one based on the functional ID of one of the entity instances involved. Remember also that it is your responsibility to clean up the log files at some point.

In a servlet, use one of the GenericServlet.log() methods to produce log application progress information, or to signal error situations. This is a way to produce logging information in a servlet container.

Run Simple Tests

Be aware that running Service Desk rule actions and Java HTTP servlets are complex subjects. While not difficult from a technical point of view, every time you install software in the context of an environment, your unfamiliarity with the environment greatly complicates your work. Therefore, before you build a rule action or a servlet, test your code in a simple standalone command-line program. If you encounter difficulties, go back to your simple command-line program. A frequent cause of problems with rule actions is that the Service Desk agent is not running, or that it was never installed.

Java Servlets

This section describes the use of Java HTTP servlets with the Service Desk Web API. An extended example of deploying on the Tomcat Web Server is also presented.

Introduction

Java HTTP servlets are small applications that live in a servlet container. Typically, a servlet container complements an HTTP web server, and the servlets serve HTTP requests that come in over the network.

The Service Desk Web API is designed to allow you to use Service Desk from Java servlets. Within this setup, there are two levels of client-server operation. The servlet is a server for the browser client but, at the same time, it is a client of the Service Desk management server by using the Web API.

The servlet container keeps a state for the servlets in the form of a session. The session is a servlet container-specific session. However, it can keep track of session-specific data for servlets. When using the Web API, the servlet container session is used to remember the Service Desk session object.

Figure 2-1   Servlets in the Web API Architecture
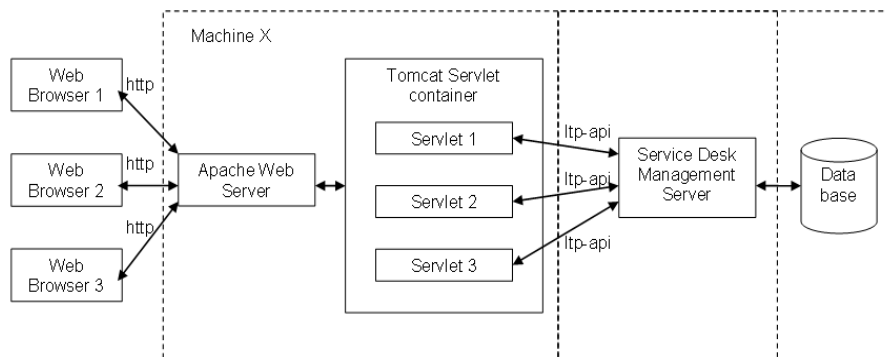
The Apache Web Server and the Tomcat Servlet Container must run on the same machine (Machine X in the figure). The Service Desk management server and the Database can be installed on the same machine, but this is not mandatory.

The examples directory contains an example of a Java HTTP servlet illustrating some of the issues involved in developing and deploying HTTP servlets using the Service Desk Web API. The servlet shows how to log in, build a session, and manage data. Refer to the documented source for more information on its use.

Examples

The `com.hp.ov.sd.webapi.servlets` package is a Java servlet that implements a simplified service call management application. After installing the example tree, the servlet source files and class files are located in
`C:\web-api\web-api-examples\com\hp\ov\sd\webapi\servlets`.
Use the `compileServlets.bat` or `compileServlets.sh` file to compile the servlet.

Deployment

This section presents an example of how to deploy a servlet on the Tomcat Web Server. For convenience, the sources are provided for you in the Examples directory. However, you will still need to modify the Tomcat configuration, and to deploy the built package.

The source files provided for this example are shown in Table 2-7.

Table 2-7   **Web API Example Files**

File	Description
`compileServlets.bat`, `compileServlets.sh`	Compile the servlet example.
index.html	Entry page for the example application. This will be included in `web-api-example.war`.

Table 2-7   **Web API Example Files (Continued)**

File	Description
`createWAR.bat,` `createWAR.sh`	Script to build the `web-api-example.war` for the example. After the `server.xml` is updated, `web-api-example.war` is the complete example web application.
`installWAR.bat,` `installWAR.sh`	Copies the `web-api-example.war` file to the correct location. The next time the Tomcat Web Server is started, the example application will work.

The Web ARchive (WAR) file containing the application-specific files now needs to be built. In this very simple example, the only application-specific files are the servlet class, an `index.html` file, the `web-api.jar` library, and a deployment descriptor file (`web.xml`).

The deployment descriptor file tells Tomcat which servlet classes correspond to which invocation URLs. The application servlets can use the deployment descriptor file to obtain application parameters. Because the example application has only one servlet class and no configurable parameters, the contents of the `web.xml` file are very simple.

Refer to the Javasoft servlet specification for information about the `web.xml` file, and the content requirements of the application `.war` file. A complete explanation of the deployment and configuration of Java Web applications is beyond the scope of this guide. It is recommended that you review the files provided with the example application. This will give you a better understanding of the Tomcat documentation. Most of the files contain references to the standard installation locations for the Service Desk Web API and Service Pages. If you have installed the software in different locations, you will need to modify the files.

To compile, package and deploy the example application, use the scripts shown in Table 2-7. Then restart Tomcat. The example application should now work. Assuming that you used the standard installation locations, the URL of the start page is http://localhost:8080/web-api-example/index.html. If you have loaded the demonstration database, a user "OLSEK" with password "servicedesk" exists. It is the caller of exactly one service call. Use the example application to enter some calls.

NOTE	This example is a practical application of an article by James Goodwill on the O'Reilly web site. For the full text, refer the original article: http://www.onjava.com/pub/a/onjava/2001/04/19/tomcat.html.

3 Changes From Service Desk 4.5 to 5.0

This chapter describes the changes between version 4.5 and version 5.0 of the Service Desk Object Model and how they impact the Web API.

Documented UI and Javaname Changes

The documents listing the object model changes are located in the `/paperdocs` folder of the Service Desk installation directory. These documents are:

- `ModelChanges4.5to5.0(UInames).html`

- `ModelChanges4.5to5.0(javanames).html`

`ModelChanges4.5to5.0(UInames).html` lists the differences between version 4.5 and version 5.1 based on class and property names as used in the Service Desk 5.0 GUI.

`ModelChanges4.5to5.0(javanames).html` lists the differences between version 4.5 and version 5.0 based on attribute names as used in the Service Desk 5.0 Java source code.

These documents do not reflect label changes, due to localization. These documents contain a reference to the Service Desk build number to which they apply.

For classes and properties that cannot be presented in the GUI, the Web API methods contain the string 'Undisplayable'. For example: for cdm entity `EMailAddress` a method exists to retrieve the address: `getUndisplayableAddressLowerCase()`.

Overview of Changes

The following changes apply:

1. Modifications to non-generated classes:

   - OIDs - introduction of a new object type

   - ApiSDSession - the session object is renamed

2. Package re-organization:

   - Packages are organized by module

3. Classpath settings:

   - Classes are distributed in multiple jar files, the names of which have changes since Service Desk 4.5

4. HP OpenView Service Desk model changes:

   - Renamed entities

   - Renamed attributes

   - Attributes removed from existing entities

   - Attributes added to existing entities

   - Attributes removed

   - Attributes added

Modifications to Non-Generated Classes

The main changes that impact existing Service Desk 4.5 Web API applications are as follows:

- A new object identification type
- Name change for the session object

Object ID

The Object ID type has changed in 5.0. In Service Desk 4.5, all Object IDs were stored as long integers (`java.lang.Long`). As of Service Desk 5.0, OID is used as the object type (`com.hp.ov.obs.OID`).

All methods that previously used Object IDs as parameters or returned Object IDs, both as long integers, now use the OID object type; long integer Object IDs are replaced by OID.

The following methods return `com.hp.ov.obs.OID`:

- `getOID()`
- `getDefaultView()`
- `getEntityID()`

The following methods accept `com.hp.ov.obs.OID` as the input parameter:

- `delete(id)`
- `findEntity(view)`
- `openEntity(oid)`
- `openNewEntity(template)`

The following example shows how to create a service call (using a template) with the OID object type:

```
// import the OID object

import com.hp.ov.obs.OID;

// Initialization code

. . .
```

```
// Create service call using template: MyExampleTemplate

ITemplate[] templates =
session.getTemplateHome().searchOnName("MyExampleTemplat
e");

// Get template OID

OID templateOID = templates[0].getOID();

//Create service call using template OID

IServicecall serviceCall =
session.getServicecallHome().openNewServicecall(template
OID);
```

Session Object

The Service Desk 4.5 Web API session object `ApiSDSession` is replaced in Service Desk 5.0 by `Session`. For example, to open a session specify the following:

```
Session session = Session.openSession(myServer,myUser,
myPass);
```

After the session is opened, you can retrieve a reference to it from any entity by calling `entity.getSession();` assuming the entity implements the `IObject` interface.

Package Reorganization

As of version 5.0, classes are packaged differently to separate classes from different modules.

In Service Desk 4.5 the Web API was distributed under the package names `com.hp.itsm.*` and `com.hp.ifc.*`. As of version 5.0, the Web API packaging is organized by module, as follows:

Table 3-1　　　**Package changes**

Package Name	Package Description
com.hp.ov.webapi.*	Web API specific entities, such as the Session object.
com.hp.ov.webapi.sd.*	Entities specifically relating to the Service Desk module/functionality in Service Desk 5.0.
com.hp.ov.webapi.obs.*	Entities specifically relating to the Object Server module/functionality in Service Desk 5.0.
com.hp.ov.webapi.slm.*	Entities specifically relating to the Service Level Management module/functionality in Service Desk 5.0.
com.hp.ov.webapi.cdm.*	Entities specifically relating to the Common Data Model module/functionality in Service Desk 5.0.
com.hp.ov.webapi.cmm.*	Entities specifically relating to the Common Metrics Model module/functionality in Service Desk 5.0.

Table 3-1   **Package changes (Continued)**

Package Name	Package Description
com.hp.ov.webapi.rpt.*	Entities specifically relating to the Reporting module/functionality in Service Desk 5.0.

NOTE   The package stem `com.hp.ov.webapi.impl.*` (not listed above) contains the implementing classes of the Java Interfaces that are contained within the packages in the above table. As a Web API developer you need the interfaces, not the implementation classes, and can therefore ignore all classes under `com.hp.ov.webapi.impl.*`.

Classpath

The Web API classes in Service Desk 4.5 are distributed in a single jar file. As of Service Desk 5.0 these classes are distributed in a set of jar files as follows:

- `OvObsWebApi-client.jar`

- `OvObsWebApi-common.jar`

- `OvObsWebApi-server.jar`

- `OvObsSDK.jar`: needed for the `OID` class

- The Web API jar file itself, containing the class files generated by compiling the generated Web API code.

This impacts the CLASSPATH setting of the Web API application.

Service Desk Model Changes

Changes to the Service Desk object model can be categorized as follows:

- Renamed entities
- Renamed attributes
- Attributes removed from existing entities
- Attributes added to existing entities
- Removed entities and attributes
- Added attributes

For the complete list of Service Desk model changes see
`ModelChanges4.5to5.0(javanames).html` and
`ModelChanges4.5to5.0(UInames).html` in the
`$installdir/paperdocs` directory.

Renamed Entities

A number of entities are renamed as of Service Desk 5.0. For example,
the Address entity is renamed PhysicalAddress. To access the version 4.5
AddressHome object you write the following:

```
IAddressHome addressHome = session.getAddressHome();
```

To access the version 5.0 `AddressHome` object you write the following:

```
IPhysicalAddressHome physicalAddressHome =
session.getPhysicalAddressHome();
```

To retrieve version 4.5 history lines for a configuration item in you write
the following:

```
IConfigurationItem[] configurationItems =
session.getConfigurationItemHome().searchOnIPAddress("127.0.
0.1");
```

```
IHistoryLinesConfiguration[] historyLines =
configurationItems[0].getHistoryLines();
```

To retrieve version 5.0 history lines for a configuration item you write
the following:

```
IConfigurationItem[] configurationItems =
session.getConfigurationItemHome().searchOnIPAddress("127.0.
0.1");

IHistoryLinesConfigurationItem[] historyLines =
configurationItems[0].getHistoryLines();
```

This logic applies to all renamed entities listed in the documents in the
`$installdir/paperdocs` directory.

Renamed Attributes

Many attributes are renamed as of Service Desk 5.0. For example, the
e-mail address attribute is renamed to PrimaryEMailAddress and the
person organization attribute is renamed to MemberOfOrganization. To
access the version 4.5 organization object for a person you write the
following:

```
IPersonHome personHome = session.getPersonHome();
IPerson[] persons =
personHome.searchOnEmail("hi@there.com");

IOrganization organization =
persons[0].getPersonOrganization();
```

To access the version 5.0 organization object for a person you write the
following:

```
IPersonHome personHome = session.getPersonHome();

IPerson[] persons =
personHome.searchOnPrimaryEMailAddress("hi@there.com");

IOrganization organization =
persons[0].getMemberOfOrganization();
```

This logic applies to all attribute changes listed in the documents in the
`$installdir/paperdocs` directory.

Removed Attributes

Many attributes are removed as of Service Desk 5.0. For example, the
EnteredByPerson attribute is now part of the Registration entity. In
version 4.5 to find the person who entered a change you write the
following:

```
// Change with ID of 54321
IChange change = session.getChangeHome().openChange(54321);
IPerson person = change.getEnteredByPerson();
System.out.println("Change was entered by: " +
person.getName());
```

As of version 5.0 to find the person who entered a change you write the following:

// Change with ID of 54321

```
IChange change = session.getChangeHome().openChange(54321);
IPerson person =
change.getRegistration().getEnteredByPerson();
System.out.println("Change was entered by: " +
person.getName());
```

This logic applies to all removed attributes listed in the documents in the `$installdir/paperdocs` **directory**.

New Attributes

New attributes are added to existing entities as of version 5.0.

This has no impact for ported Web API applications.

Removed Entities

Some entities are removed as of Service Desk 5.0. Any code that references these entities must be re-written to reference the equivalent entities.

See `ModelChanges4.5to5.0(javanames).html` **and** `ModelChanges4.5to5.0(UInames).html` **in the** `$installdir/paperdocs` **directory**.

Changes From Service Desk 4.5 to 5.0
Service Desk Model Changes