
Parallels

APS 1.2 Controller Implementation

Developer's Guide

Revision 1.0.12



(c) 1999-2009

Contents

Introduction	4
About Application Packaging Standard	4
About This Guide	4
Useful Links	6
Typographical Conventions	6
Feedback	7
Concepts	8
Controller Perspective	9
Functions	12
Implementing Operations on Application Packages	13
Adding and Removing Application Packages	14
Possible Scenario	17
Configuring Application Global Settings	19
Possible Scenario	20
Implementing Operations on Application Instances	22
Provisioning Application Instances	24
Deploying Application Files	25
Processing Service License	25
Determining and Satisfying Requirements	25
Installing Service License	27
Processing Settings	28
Provisioning Service	30
Possible Scenario	33
Removing Application Instances	36
Possible Scenario	36
Updating Application Instances	38
Resolving Update Packages	39
Patching and Upgrading an Instance	40
Possible Scenario	42
Configuring Application Instance	45
Possible Scenario	46
Enabling and Disabling Service	48
Possible Scenario	49
Application Data Retention	50
Backing Up Service Instance	50
Restoring Service Instance	51
Migrating Service Instance	51
Possible Scenario	51
Getting Resources Usage Report	53
Possible Scenario	54
Appendix. Sample APP-META.xml	55

CHAPTER 1

Introduction

The Application Packaging Standard is an application packaging format designed to help implement a Software-as-a-Service (SaaS) business model.

This chapter provides general information about Application Packaging Standard and this guide.

About Application Packaging Standard

Because of the extensive development and integration requirements, hosting providers today offer a limited variety of hosted applications. As a result, independent software vendors have little incentive to create hosted applications that they cannot sell, and customers suffer from limited application choices.

In response to this, *Application Provisioning Standard (APS)* was developed. It is an open format of packaging and managing web applications that will make it easier for the whole hosting industry to take advantage of the expanding software-as-a-service (SaaS) market.

By defining such open format, APS increases business opportunities for the entire hosting ecosystem bringing together application vendors and hosting service providers. By implementing APS, application vendors get access to a vast sales and marketing channel of APS-enabled hosting providers. In turn, hosting services providers, by implementing APS, gain access to a great variety of APS applications.

APS implemented by a hosting services provider means that the provider uses such software that processes APS applications in full compliance with the rules defined by the standard. *The Controller*, to which this document is devoted, is exactly that part of the software which is responsible for such processing.

About This Guide

The purpose of this document is to provide guidelines for implementing *Application Provisioning Standard (APS)* on the side of hosting services providers.

The document is intended for developers of hosting provider software who want to develop *APS engine*, or *Controller*, - a part of hosting provider software processing APS site applications in full compliance with the rules defined by the standard.

Abbreviations, Definitions and Conventions

- *APS* is used instead of Application Packaging Standard in some long sentences where using it will not change the meaning of the sentence.

-
- *Hosting Environment* is an environment that provides resources or equipment needed to host a service. For example, web site and virtual machine.
 - *Web site* is a collection of web pages, typically common to a particular domain name or sub-domain on the World Wide Web on the Internet.
 - *Control panel* or *CP* is hosting provider software designed for managing user accounts, web sites and site applications.
 - *Site application* or *application* is a Web application that can be hosted on a web site.
 - *Application Package* or *Package* is a site application in its distribution format which includes all application and application-related files created, structured and packed according to the APS.
 - *Application Instance* is a site application installed from application package on a particular web site and accessible via a unique URL.
 - *Controller* is APS engine, a control panel part responsible for processing application packages and instances.
 - *Aspect* is an additional specification that declares how to describe in the application package metadata: file technologies used by an application, application management scripts, and how these descriptions must be processed by Controllers. The technologies include any software or executable code.
 - *Qualified technology* means any software or executable code described in an aspect included in the APS: Package Format Specification.
 - *User* is any CP user who has access to the Controller's functionality.
 - *Repository* is a Controller's structural part capable of storing information about all packages that can be instantiated by the Controller.
 - *Configuration* is a set of application settings values, in whichever format it is presented in a Controller implementation.
 - *XPath* notation is used to describe the structure of XML documents and to refer to XML elements and attributes in the text.
 - *APS: Package Format Specification* is referred to as *the Specification*.

Useful Links

Official APS site. Latest news and relevant information on Application Packaging Standard development.

<http://www.apsstandard.com/>

APS: Package Format Specification

<http://www.apsstandard.com/r/doc/package-format-specification/index.html>

APS: Application Packaging Developer's Guide

<http://www.apsstandard.com/r/doc/aps-packaging-dev-guide/index.htm>

XML Technologies:

XPath Homepage

<http://www.w3schools.com/xpath/default.asp>

RELAX NG Compact Syntax Tutorial

<http://www.relaxng.org/compact-tutorial-20030326.html>

Typographical Conventions

Before you start using this guide, it is important to understand the documentation conventions used in it.

The following kinds of formatting in the text identify special information.

<u>Formatting convention</u>	<u>Type of Information</u>	<u>Example</u>
Special Bold	Items you must select, such as menu options, command buttons, or items in a list.	Go to the System tab.
<i>Italics</i>	Titles of chapters, sections, and subsections. Used to emphasize the importance of a point, to introduce a term or to designate a command line placeholder, which is to be replaced with a real name or value.	Read the Basic Administration chapter. The system supports the so called <i>wildcard character</i> search.
Monospace	The names of commands, files, and directories.	The license file is located in the http://docs/common/licenses directory.

Preformatted	On-screen computer output in your command-line sessions; source code in XML, C++, or other programming languages.	<pre># ls -al /files total 14470</pre>
Preformatted Bold	What you type, contrasted with on-screen computer output.	<pre># cd /root/rpms/php</pre>
CAPITALS	Names of keys on the keyboard.	SHIFT, CTRL, ALT
KEY+KEY	Key combinations for which the user must press and hold down one key and then press another.	CTRL+P, ALT+F4

Feedback

If you have found a mistake in this guide, or if you have suggestions or ideas on how to improve this guide, please send your feedback using the online form at <http://www.parallels.com/en/support/usersdoc/>. Please include in your report the guide's title, chapter and section titles, and the fragment of text in which you have found an error.

CHAPTER 2

Concepts

This chapter describes the general factors that affect the Controller and its requirements. This chapter does not state specific requirements. Instead, it provides a background for the specifications defined in detail in the next chapter, and makes them easier to understand.

The Controller Perspective section focus is the Controller as the part of Web Hosting Control Panel software. The section defines the important Controller parts and how they are connected with other CP components.

The Functions section provides a summary of major functions that a Controller should perform.

Controller Perspective

Controller is a part of a larger system - hosting services provider's control panel. In the view of Controller's successful functioning, it should be able to interact with other CP's components, which can be generally outlined as follows:

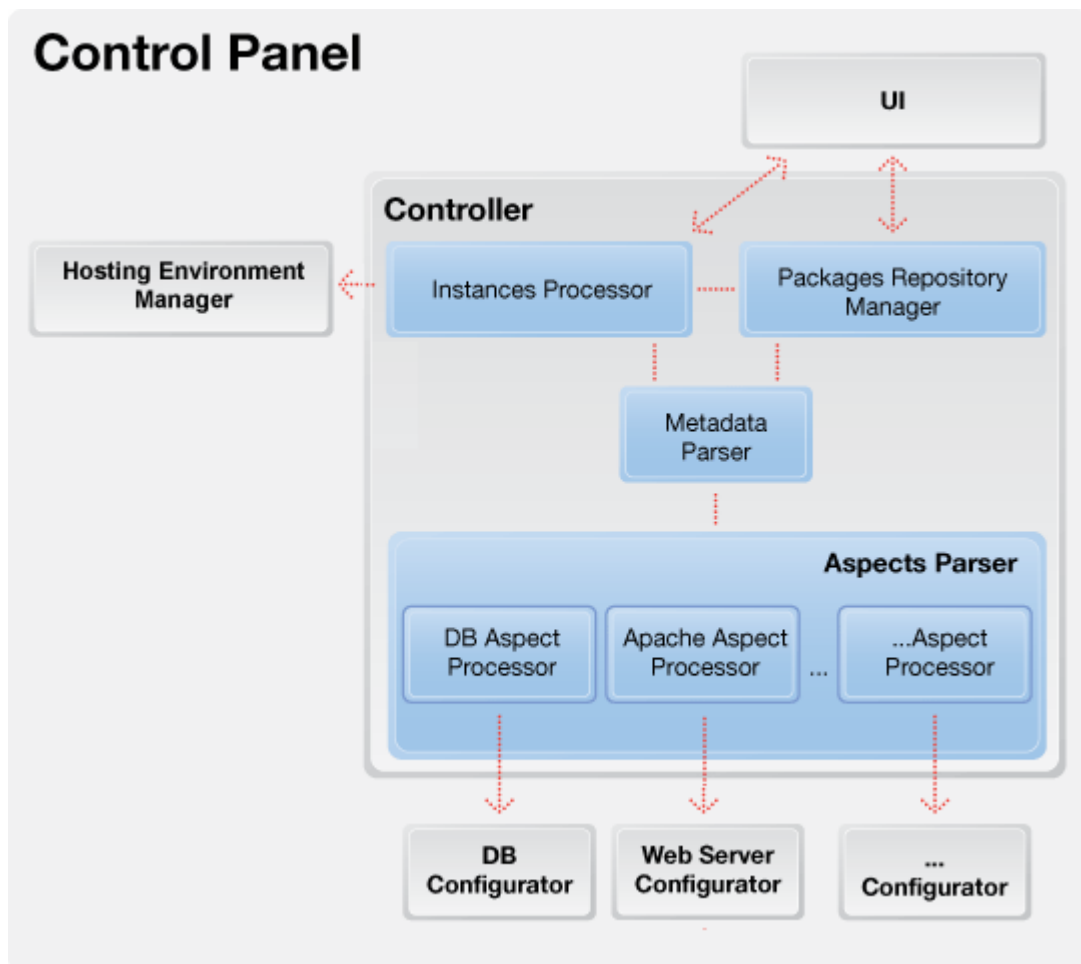


Figure 1: Controller perspective

For better understanding Controller, we distinguish the following Controller's functional parts:

- Packages Repository Manager
- Instances Processor
- Metadata Parser
- Aspects Parser

Packages Repository Manager

Storing information about application packages that can be instantiated is the obligatory condition of the whole Controller functioning. We call such 'storage' of packages metadata a *Repository*, regardless of what form it can take in a particular Controller implementation. What's important is that a Controller must have such structural part capable of storing information about all instantiatable packages.

Besides information on instantiatable packages, the Repository should also store the following:

- General information about existing package instances (it must at least be info on what application instantiated on which site, and at which URL it is accessible).
- Application global configurations.

We call the Controller's functional part performing any operations on the Repository level the *Packages Repository Manager*. We refer to it when talking about actions that a Controller performs on application packages, as opposite to actions on application instances. Therefore, Packages Repository Manager must interact with control panel user interface. Refer to the Functions section (on page 12) for details on Controller actions.

Instances Processor

We call the Controller's functional part performing any operations on the hosting environment level the *Instances Processor*. We refer to it when talking about actions that a Controller performs on application instances, as opposite to actions on application packages performed by Packages Repository Manager. Refer to the Functions section (on page 12) for details on Controller actions.

Instances Manager should store the following information:

- Local configurations of application instances;
- Information about requirements satisfied for each application service;
- Information about resources allocated to application instances (databases, virtual hosts, etc.) and changes made to external systems (e.g., web server configuration).

Instances Processor must interface into the hosting environment part responsible for managing sites and virtual machines, which we called Hosting Environment Manager just to designate the function. A Hosting Environment Manager is supposed to provide all control panel functionality related to hosting sites: creating and removing virtual hosts, setting limits on system resources used by hosts, enabling, configuring and disabling various services on sites, and so on.

Metadata Parser

Package Metadata Parser is the obligatory functional component of a Controller capable of parsing package metadata files, understanding and validating them in accordance with the rules of basic metadata format defined by the Specification, and then passing it for the following processing. The basic format of a package metadata file `APP-META.xml` is defined by the Specification in a set of RELAX NG XML schema files which should be used by Metadata Parser.

Metadata Parser is involved in all Controller operations and is invoked by either Packages Repository Manager or Instances Processor. This Controller component doesn't need any interactions with any control panel parts.

Aspects Parser

Aspects Parser is another Controller's obligatory functional component which is capable of analyzing aspects-specific data contained in the `APP-META.xml` files and passing it for the following processing back to Instances Processor, Repository Manager or a particular technology configurator. This is the data about application requirements, files handlers and permissions, and the configuration script interpreters. The Aspects Parser operates basing on the rules of data format defined by aspects.

Aspect is an additional specification that declares how a web application's specific need (technology) must be described in packages, and how these descriptions must be processed by Controllers. The aspects common for the most of web applications, such as PHP aspect, ASP.NET aspect, database aspect, etc., are included in the Specification in section 7. Common Aspects.

Aspects are considered as an open and extensible part of the Specification, since an application can use technologies other than those described by the common aspects. If an application uses such technologies, each must be defined by the corresponding additional aspect. So, when creating the Aspects Parser, consider making its 'knowledge base' easily expandable, considering that, later on, you Controller might deal with the packages that use some additional, unknown aspects, which must be painlessly and quickly added to the Aspects Parser's knowledge base.

Functions

The major functions that a Controller performs are the following:

- *Adding application package to Repository.* Application package is validated, recognized, and its information is stored in the Repository, what makes instantiating the package possible.
 - *Configuring application global settings.* After application package is added to the Repository the Controller must configure its global settings. Global settings are configured on the Repository level beyond operations on the application instances, and are stored to be applied at the moment of the package instantiating. Global settings, as soon as saved, are applied to all the application instances.
- *Provisioning application instance.* Application package is instantiated in a hosting environment, meaning that a working application instance can be accessed through web. This operation always includes *at least* 'root' services provisioning.
- *Provisioning application instance services (services instances).* Services instances are provisioned according to hierarchic structure and provisioning methods defined in application metadata file.
- *Configuring service instance settings.* Services settings are configured on the instances level, at the moment of creating an instance. Some of the settings are reconfigured later affecting the application instance.
- *Enabling/Disabling service instance.* If the ability to change service status is declared for configuration script specified for this service, service status can be changed to *enabled* or *disabled* respectively. Disabled service functions are not available for service owner.
- *Updating application instance.* If application package of a newer version and/or release is available in Repository, application instance is updated by it, meaning that the old application files are replaced with the new ones so that the updated application provides all the new functionality with all the user-created files and configurations from the previous version preserved. Depending on the update status - upgrade or patch - the operation is performed with or without user's attendance. Updating application instance comprises updating its services instances. Necessity of updating particular service instance depends on provisioning method specified for it.
- *Removing application instance services.* Service instance files are removed and respective allocated resources are freed.
- *Removing application instance.* Application instance is uninstalled from hosting environment. This operation comprises service instances removal and deallocating hosting environment resources used by application services.

CHAPTER 3

Implementing Operations on Application Packages

This chapter provides information on the constraints that the Specification imposes on the implementation of Controller's functionality.

The chapter focus is a Controller's function performed on application packages. To make the Controller implementation requirements stated in the chapter clearer, we illustrated each function-related statement with a sample scenario of how the function can be implemented.

Operations on application packages include the following:

- Adding/Removing application packages to/from Repository;
- Configuring application package global settings.

Adding and Removing Application Packages

This section considers adding and removing application packages to/from Repository.

Adding Application Packages

Performing adding package operation correctly is very important as it predetermines further successful package processing. Before adding a package to Repository it should be validated. The validation procedure may comprise the following steps:

- 1 *Validating basic conformity.* The Controller must first validate the package basing on the following criteria:
 - the package file has the `.app.zip` extension;
 - the package is successfully unzipped;
 - the package contains an `APP-META.xml` file in its root;
 - the `APP-META.xml` is parsed by the Metadata Parser according to the rules of basic metadata format defined by the Specification.
- 2 *Recognizing a package.* The most important point about adding an application package is that the Controller should accurately recognize it. The Controller actions to be performed for this purpose differ depending on APS format version the application is packaged with. So, the first step is defining APS format version. The Controller should get APS format version from the package metadata file with Metadata Parser. As defined by the Specification, this data is contained as follows:

```
<application xmlns="http://apsstandard.com/ns/1" version="1.1">
```

Note: if the `version` attribute is absent, the APS format version is 1.0.

- *APS 1.0.* Metadata Parser successfully gets the application name, version, and release from the `APP-META.xml` file. It is enough for the Controller to know this general information about the package. As defined by the Specification, this data is contained in the metadata file as follows:

```
<name>phpBB</name>
<version>2.0.22</version>
<release>6</release>
```

- *APS 1.1.* Metadata Parser successfully gets the application name, version, and release from the `APP-META.xml` file. It is enough for the Controller to know this general information about the package. As defined by the Specification, this data is contained in the metadata file as follows:

```
<name>phpBB</name>
<version>2.0.22</version>
<release>6</release>
```

- *APS 1.2.* This APS format version involves the following criteria to be validated by the Controller:
 - the package contains a `APP-LIST.xml` file in its root;
 - the `APP-LIST.xml` is parsed by the Metadata Parser according to the rules of contents listing format defined by the Specification;

- Metadata Parser successfully gets list of files and their corresponding SHA256 values from the `APP-LIST.xml`;
- the Controller checks that the list of package files and their current SHA256 values match those extracted from the `APP-LIST.xml`. This operation ensures package integrity;
- Metadata Parser successfully gets list of package signatures, if any from the `APP-LIST.xml`;
- the Controller checks signatures validity to ensure package origin;
- Metadata Parser successfully gets the application unique identifier, the application name, version, release and master package identifier (if any) from the `APP-META.xml`. As defined by the Specification, this data is contained in the package metadata file `APP-META.xml` as follows:

```
<application xmlns="http://apsstandard.com/ns/1" version="1.2">
  <id>http://www.phpbb.com/</id>
  <name>phpBB</name>
  <version>2.0.22</version>
  <release>6</release>
  <master-package>
    <package id="http://www.phpbb.com" match="version = 1.1"/>
  </master-package>
  ...
</application>
```

If the Controller gets the identifier of a master package from the package metadata, it must check whether the master package is in Repository already or not. If the master package is in Repository the Controller should somehow store the information about their master to add-on relations. Otherwise, the Controller must refuse to add the package to Repository returning the proper error message.

The unique application identifier is very important for further package processing. The Controller must ensure that this identifier is really unique for an application among others in Repository.

Note: for details on the metadata file format, refer to the 5. Metadata Descriptor section of the Specification.

- 3 *Displaying validation result.* If a package successfully passes validation procedure the Controller may display a success message to a user. If a package fails to pass the validation basing on any criterion above, the Controller must refuse to add the package to Repository returning the proper error message.

These criterions of package validity are not the only ones, but the ones required on the stage of adding a package to Repository so that the package then can always be uniquely identified by Controller. The other package validity criteria are defined by the Specification in the section 4. Basic Package Format.

We recommend that when adding a package to Repository, the Controller checks whether it can recognize the metadata elements defined by aspects, namely, the elements which describe application requirements, provision methods (URL handlers and/or configuration script language) and permissions. This is the best way to detect non-qualified technologies used by the application as early as possible, and to expand the Aspects Parser functionality in a good time.

Removing Application Packages

It's worth considering the implementation of removing packages from your Repository. Note that it is only a recommendation, there are no rules for removing packages defined by the Specification, so this Controller operation is at your option.

Possible Scenario

Conditions.

The Controller is implemented so that its Repository has a form of physical directory where all application packages are stored unzipped, each in its own specific sub-folder.

The Repository Manager stores all information about application packages in the database. The info includes path to a package in the Repository, all package metadata from `APP-META.xml`, package global configuration, information about instances, and so on.

The Repository Manager has its representation in CP GUI which allows CP users to upload application packages to the Repository, see the results of upload, see a list of packages in Repository.

Starting Point. A user initiates uploading an application package via GUI, using the standard uploading form at the Repository manager screen.

Step 1. The Controller verifies that the offered file has the `.app.zip` extension and allows uploading it to some upload directory. In case the file has a wrong extension, the Controller refuses the file upload and returns the appropriate error message to the user.

Step 2. The Controller unzips the uploaded file to the temporary directory. If unzipping fails, the Controller deletes the uploaded file and returns the appropriate error message to the user.

Step3. The Controller searches for `APP-META.xml` among the extracted files and validates its basic format:

Sub-step 3.1. The Controller searches for `APP-META.xml` among the extracted files and starts reading it with the Metadata Parser. If the metadata file is missing, the Controller deletes the package files (both the archive and the unzipped files) and returns the appropriate error message to the user.

Sub-step 3.2. The Metadata Parser gets the information about APS format version the application is package with. Let's assume that `version` attribute is absent, so the application is packaged with APS 1.0.

Sub-step 3.3. The Metadata Parser gets the information about application name, version and release. If any is missing, adds a message about it to error stack.

Sub-step 3.4. The Metadata Parser reads the rest of the metadata file, and checks if it complies with the RELAX NG schema of basic metadata descriptor. If a required element is not validated according to the XML schema, the Controller adds a message about it to error stack. If an optional element is not validated, it is skipped and then treated as it does not exist.

Sub-step 3.5. The Controller checks if the package contains scripts in the `/scripts/` folder. If it does, the Controller checks if the package metadata contains the declaration of configuration scripts language. In case when there is a configuration script without its language declaration, the Controller adds a message about it to the error stack.

Sub-step 3.6. The Controller checks the state of error stack. If it contains any messages, the Controller deletes the files and displays a *Package Could Not Be Validated* screen with the content of error stack. If the error stack is empty, the Controller continues to the next step.

Step 4. The Controller checks with the Aspects Parser if it recognizes all the aspects-related data required for instantiating:

Sub-step 4.1. Reads and recognizes the application requirements (content of the "requirements" elements).

If meets an unknown

`"/application/service/requirements/<requirement>"` element, adds a message about it to the error stack.

If meets an unknown

`"/application/service/requirements/choice/requirements/<requirement>"` element, marks the parent element

`"/application/service/requirements/choice/requirements"` as

Unknown. If all children of a `"/application/service/requirements/choice"` element are Unknown, adds a message about it to error stack.

Sub-step 4.2. Reads and recognizes the application URL mapping rules (content of the `"/application/service/provision/url-mapping"` element). If meets an unknown element, adds a message about it to error stack.

Sub-step 4.3. Recognizes the declaration of the configuration script language (value of the `"/application/service/provision/configuration-script/script-language"` element) if any presents in the package metadata. If the language is unknown, the Controller adds a message about it to error stack.

Step 5. The Repository Manager adds the package to the Repository:

Sub-step 5.1. In the Repository folder, creates a package-specific directory using for its name the application ID, version and release and moves the package files from the temporary location to this folder.

Note: application package name is used instead of ID in APS earlier versions.

Sub-Step 5.2. Adds all information about the package to the Repository database.

Sub-Step 5.3. Checks the state of error stack. If it is not empty, the Controller flags the package as yet-non-installable and saves the error stack messages to a file (to be used during the following expanding of the Aspects Parser).

Sub-Step 5.4. Displays to the user a Repository screen with the list of packages (the target package highlighted) and the message about operation results: either that the package is successfully added or, in case error stack was not empty, with the error stack content.

Configuring Application Global Settings

The Controller should implement mechanism of configuring application global settings.

Application global settings are used to:

- Set up general application configuration *before* instantiating the package, so that when the application instance is created, this configuration is automatically applied.
- Change application configuration that affects all the application instances, as soon as it is changed.

These use cases imply the following requirements for the Controller implementation:

- The Controller must store a package global configuration in order to be able to use it each time the package is being instantiated.
- The Controller must distinguish the global settings in order to initiate reconfiguring application instances when these settings are changed.

Application global settings information is contained in the basic package metadata. Such settings are declared by the `global-settings` element within the application description, in the format defined by the Specification in the section 5.1.17. Global Settings. The global settings that must not be involved in reconfiguration of existing application instances are marked with the `installation-only="true"` attribute.

Application global settings may require uniqueness of their values within some scope. Such settings are marked with `uniq` attribute that value defines scope of setting value uniqueness. The Controller must validate and ensure uniqueness of such settings values. For details, refer to the 5.2.5 Service Settings section of the Specification.

Values of application global settings may be verified with special verification script declared in the `/application/provision/verify-script` element. If verification script is declared, the Controller may run it itself or provide a user with dialog window where user can click special control for verification, for example, Verify button. A verification script definition may include `structured-output` element. In this case a script returns report on erroneous values in the form defined by the Specification. The Controller should catch output stream independently of error stream and process it. Also, a verification script may return list of choices for `enum` setting type, it got from an application. The Controller should cache this list of choices and suggest it for respective setting editing. For details on verification script, refer to the 5.3.2.3 Configuration script output and 5.3.3 Verification Script sections of the Specification.

Possible Scenario

Conditions.

The Control Panel is implemented so that it allows user to open a screen with an application Global Configuration form where the user can specify global settings of the application.

The Controller is implemented so that it stores all instance-specific data in the database. The data includes a copy of `APP-META.xml` file, information about the instance specific URL, settings, and so on.

Starting Point. User initiates opening the Global Configuration screen of an application.

Step 1. The Controller gathers all information necessary for generating the form:

Sub-step 1.1. Finds the required application package, and reads the settings info from the basic metadata selecting descriptions of global settings ("`//global-settings`")

Sub-step 1.2. Processes the global settings info (setting name, description, data type, default value, grouping information), and makes the form skeleton - the global settings grouped as specified in the metadata.

Sub-step 1.3. Finds the saved values of global settings and fills them in the form skeleton. If finds no previous configuration, uses the default values from the package metadata.

Step 2. The Controller transforms the form skeleton to the form code using the set of rules that define how the setting information must be reflected in GUI, and displays the Global Configuration screen.

Step 3. After user fills in the Global Configuration form and submits it, the Controller processes the submitted data:

Sub-step 3.1. Validates the submitted data using the settings type information and for those settings that have `uniq` attribute ensures their values uniqueness. If the validation fails, the Controller displays the configuration screen form where the settings that failed validation are marked, then goes to the beginning of the **Step 3**.

Sub-step 3.2. Saves the global settings.

Sub-step 3.3. Selects all settings elements which lack the `installation-only="true"` attribute, and prepares for each such setting an environment variable `SETTINGS_<id>`, where `<id>` replaces the value of the setting's `id` attribute ("`//setting[@id]`").

Step 4. The Controller reconfigures all instances associated with the package according to the new global settings. It goes through the list of the instances, and performs the following actions with each instance:

Sub-step 4.1. Reads the instance metadata, particularly, the info on instance URL, settings, satisfied requirements, etc., and prepares all possible environment variables.

Sub-step 4.2. Runs the application configuration script in the environment where all the environment variables prepared on the previous sub-step are defined.

Sub-step 4.3. As the script returns an exit code, the Controller proceeds to reconfiguring the next instance (goes to **Sub-step 4.1.**). If the script invocation fails, the Controller adds a message about it to the error stack.

Note: For details on invocations of application configuration scripts, refer to the Implementing Operations on Application Instances (on page 22) and Configuring Application Instances (on page 45) sections in the current document, and section 5.3.2. Configuration script in the Specification.

Step 5. As soon as the Controller finishes processing all instances, it displays the operation results screen adding to it the error stack content.

CHAPTER 4

Implementing Operations on Application Instances

This chapter provides information on the constraints that the Specification imposes on the implementation of Controller functionality.

The chapter focus is a Controller functions performed on application instance. To make the Controller implementation requirements stated in the chapter clearer, we illustrated each function-related statement with a sample scenario of how the function can be implemented.

Starting APS 1.1 an application instance comprises services. Services are an application functions or features that are delivered to end-user. Application may provide independent services as well as nested ones. Independent services are, so called, application 'root' services. Application must contain a 'root' service. A 'root' service may declare a number of child services. Each application service may be provisioned a number of times. Operations on services instances are performed within application instance.

The following table lists operations available in APS. Operations that can be performed on application instance or on service instances within installed application are marked with "X" in respective column.

Operation	Application	Service
Provisioning	X	X
Removing	X	X
Configuring	X	X
Backing up	X	X
Restoring	X	X
Getting resources usage report	X	X
Enabling/disabling		X
Patching	X	
Upgrading	X	
Migrating	X	

Important Points

One should pay attention to the issues below during implementing Controller functionality.

Archive content processing

Application may require a special processing of application archive content prior to its actual provisioning. Such requirements are declared with `application/content` element in the package metadata. For details, refer to the 5.1.16 Content Delivery Methods section of the Specification.

Services handling

Since an application instance comprises services the following information is specified for each service individually:

- presentation information;
- info-links to external web resources;
- entry points;
- settings;
- requirements to be satisfied;
- end-user license agreement;
- operations available.

Child services may refer to requirements and settings of their parent one or declare their own.

All mentioned operations are performed on each application service individually. Operations on child services are performed within their parent one. Some application services can be provisioned automatically without a user involvement using settings default values. Whether service is to be provisioned automatically or not completely depends on Controller implementation.

Scripts handling

Usually, application package contains configuration scripts for each application service declared. Scripts should be located at the `/scripts/` folder in the package root. The Controller must invoke respective script when performing any operation on application service, as defined by the Specification in section 5.3.2. Configuration Script. A configuration script may be used for provisioning, configuring, upgrading, removing, enabling and disabling application services.

A configuration script invocation means that the Controller runs the script with a particular interpreter in a specific environment where all the necessary data is presented in a form of environment variables. The Controller must choose the interpreters basing on the information about programming language in which each script is written. This info is provided by the `//script-language` element in the package metadata defined by the Specification in section 5.3.2. Configuration Script. Binary executable scripts are allowed and declared with `//binary-executable` element.

The Controller must always pass to the configuration script all the information about the application it could get and transform to environment variables for every script invocation, no matter what the invocation purpose is. The environment variables passed to configuration scripts are regulated by the Specification in section 5.3.2.2. Environment Variables. For better understanding what environment variables must be created by a Controller and then passed to application configuration script, refer to the Appendix. Sample APP-META.xml (on page 55).

An application may additionally contain scripts that serve different from configuration one purposes, for example settings verification, resource usage reporting, application license installation, backing application up.

The details of Controller behavior specific for invoking scripts during a particular operation are considered further in the chapter.

Provisioning Application Instances

Generally speaking, the Controller should do the following when instantiating an application package:

- 1 Prepare all the data and conditions necessary for instantiating:
 - Determine application content delivery methods (if any) and prepare a hosting environment accordingly. See the section 5.1.16 Content Delivery Methods in the Specification.
 - Retrieve the application's global configuration and be ready to apply it to the instance. (See the 5.1.17. Global Settings, 5.2.5. Service Settings sections in the Specification, the Configuring Application's Global Settings (on page 19) section.

The order of these preparatory sub-stages is at discretion of implementer.

- 2 Deploy application files to a hosting environment. See the section 5.1.16 Content Delivery Methods in the Specification.
- 3 Provision application instance services.
 - Determine application 'root' service to be provisioned.
 - Determine selected service hierarchy and order of provisioning accordingly.
 - Present service-usage license text (if any is provided in the package) to a user and provide tools necessary for accepting the license if it is required. See the 5.2.2 License Agreement section in the Specification.
 - Determine and satisfy the application service requirements. This step includes application license installation, if it's required. See the 5.2.7. Requirements, 7.4. Environment Variables sections in the Specification.
 - Prompt the user to configure the application service installation settings and process the submitted settings. See the 5.2.4. Entry points, 5.2.5. Service Settings, 7.4. Environment Variables sections in the Specification.
 - Provision service to a hosting environment. See the 5.3. Service Provision Methods section in the Specification.

- Run the service configuration script passing to it all the available information about the service in the form of environment variables created on the previous stages. See the 5.3.2. Configuration script, 7.6. Configuration Script language, 7.4. Environment Variables sections in the Specification.
- Determine child services and perform the same steps as for their parent one, mentioned above in the step 3. But at first, the Controller must check whether the child service is declared as singular (possess `singular="true"` attribute). If the attribute is present the Controller must check that this service is not already installed under its parent. Otherwise, it is allowed to create several instances of the service.

Further in the section, we will consider each part of the instantiating process in detail.

Deploying Application Files

Deploying application files to the site depends on content delivery method declared with `"application/content"` element in the application metadata file, for example Parallels Virtuozzo Container template. Allowed content processing options are defined in aspects. If no content delivery method declared the Controller should use the default one. Default content delivery method implies copying the very application files and directories from the package to a hosting environment specific location.

Processing Service License

Information about the service usage license is defined by the `"/service/license"` element in the basic package metadata (`APP-META.xml`). The element format is defined by the Specification in section 5.2.2 License Agreement.

If the `"/service/license"` element has the `must-accept="true"` attribute, the Controller should accompany displaying the license text with a user dialogue which enables the user to accept the license terms, and continue provisioning the service only if the license is accepted.

Determining and Satisfying Requirements

An application service may have specific hardware and software requirements, which must be recognized and satisfied by the Controller before it instantiates the service.

Hardware Requirements

Application services may require the following hardware resources:

- *Diskspace*. The amount of free hard disk space necessary for the application service to be deployed in a hosting environment. This requirement is defined by the optional `"/service/provision/url-mapping/installed-size/"` element in package metadata file. If the package metadata contains the element, the Controller must check that the target environment has enough disk space for installing the application service.

- *Server CPU and RAM.* The server CPU frequency and size of memory necessary for the application service to work. These requirements are defined by the attributes of the optional `/service/requirements/hw` element in package metadata file. The element may declare the minimum and recommended amount of CPU and RAM resources required. If the package metadata contains the element, the Controller must check that the target environment has at least minimum amounts of CPU and RAM resources for installing the application service. The element format is defined by the Specification in section 8.6. Hardware Resources.

Software Requirements

The application service software requirements are defined by the `/service/requirements` element in the package metadata file `APP-META.xml`. The element format is defined by the Specification in section 5.2.7 Requirements.

Generally speaking, application service requirements are a collection of statements that a Controller must satisfy before deploying the service. Each element nested in `/service/requirements` states a particular requirement, which can be done in the following two ways:

- simple statement describing one requirement which must unconditionally be satisfied (hereinafter referred to as *requirement element*)
- complex statement that represents at least two equivalent requirements combinations, one of which must be satisfied (hereinafter referred to as *requirement choice*)

Ultimately, a complex statement is a set of simple statements organized in a specific way. The way is as follows: a complex statement itself is defined by the `/service/requirements/choice` element. The requirements combinations stated within are defined by the `/service/requirements/choice/requirements` elements, each having the obligatory attribute `id` uniquely identifying this combination of requirements. And the `/service/requirements/choice/requirements` child elements are, again, requirement elements.

Each requirement belongs to the one of requirement types. These types are defined in the aspects and dictate the format and meaning of requirement elements as well as rules of how to satisfy the requirement during instantiation. For details, refer to the 8. Common Aspects section in the Specification.

To determine application requirements, the Controller must do the following:

- Get information about requirement elements and requirement choices by parsing the `APP-META.xml` file with the Package Metadata Parser.
- Process the requirement elements and requirement choices data with the Aspects Parser.

The Controller must proceed to satisfying requirements only after it determines all of them. When dealing with requirement choices, the Controller should act as follows: If a requirement choice contains at least one `/application/requirements/choice/requirements` element where all requirements are determined, the Controller recognizes this requirement choice as determined and continues to satisfying requirements.

The Controller satisfies service requirements basing on the rules defined in the corresponding aspects. The Common Aspects included to the Specification define for each requirement type exact conditions of its satisfaction, and what must be done upon this satisfaction, in particular, what environmental variables it must create to pass further to application configuration script.

The Controller must continue to the next stage of instantiating application service only if all the application service requirements are satisfied, which implies that all simple statements are satisfied, and all complex statements have at least one requirements combination satisfied. As soon as the Controller satisfies the requirements, it must prepare the following environment variables:

- those defined in aspects as must-be-returned for particular requirement types.
- those indicating the satisfied requirements combination within each requirement choice (in the format `CHOICE_<id>`, for details refer to the 5.2.7. Requirements section in the Specification).

The Controller must store information on requirements being satisfied for parent service as it must be available for provisioning of its child services. The set of environment variables passed to configuration script of child service must also contain variables with information about satisfied requirements of parent service. Requirements declared by child service override values of propagated variables.

On the stage of processing service requirements it may be necessary to allocate system resources for the service needs (e.g., a database). In such cases, also defined by aspects, the Aspects Parser returns information about the resources allocated in the form of environment variables. It is important that the Controller stores information about the allocated resources in order it could deallocate them when removing the application instance.

Installing Service License

An application service may require license installation in addition to or instead of end-user license agreement. Licensing of APS applications is performed by means of license keys. Getting license keys involves arranging communication between the Controller or Control Panel and license authority, for example Parallels License Administrator.

Generally the process of license installation is the following:

- 1 The Controller gets information about license requirement elements (`"/service/requirements/l:license"`) by parsing the `APP-META.xml` file with the Package Metadata Parser.
- 2 The Controller gets information about license manipulation script definition (`"/service/provision/license-script"`) by parsing the `APP-META.xml` file with the Package Metadata Parser.
- 3 The Controller process the requirement elements data and script definition with the Aspects Parser.

- 4 It may be declared that for license installation an application should be queried first to get activation data (`l:need-query` element is present). In this case, the Controller runs license script with the `query` argument, passing to it all required environment variables. Then checks whether script output matches activation data XML schema. If script output is correct, the Controller passes activation data to license authority directly or through Control Panel and proceeds with the application instance deployment.
- 5 It is not guaranteed that license will be provided to the application immediately after the installation. The Controller should ensure that access to application without license installed is set up according to policy provided in the license declaration. Policy access is defined by presence or lack of the `l:optional` element. If the element is present an application may be provided to end-user without valid license. Otherwise, an application becomes available to end-user only after the license installation.

For details on license installation terms and conditions, refer to the Licensing Aspect.

Processing Settings

Configuring Installation Settings

On this stage of instantiating service, the Controller should prompt a user to configure parameters needed for service installation and configuration:

- *Full instance URL*. The full address at which the installed application will be accessible. The option is prompt, in case, URL Mapping is declared as service provisioning method. The form in which the full instance URL is prompted is not defined in the Specification and completely depends on the Controller implementation. What is important here is that the URL data input by user is correctly validated and processed by the Controller so that this results in successful creation of the environment variables - `BASE_URL_SCHEME`, `BASE_URL_HOST`, `BASE_URL_PORT`, `BASE_URL_PATH` - defined by the specification. For details, refer to the 5.3.2.2.1. URL Mapping Variables section in the Specification.
- *Service settings*. Additional parameters needed for successful service provisioning and configuration. The options are prompt, in case, service declares settings and their values can not be determined without user intervention. Selection of settings that should be prompted depends on the Controller implementation.

Service settings are described in the package metadata file `APP-META.xml` with `"/service/settings/setting"` elements.

When prompting the user to configure the service settings, the Controller must do the following:

1. Select settings to be prompted. The Controller may rely on setting attributes, such as:
 - `class` - attribute informs about the setting or group of settings meaning. For details, refer to the 5.2.5.2 Settings Semantics section in the Specification;
 - `value-of-setting` - attribute defines value of what setting should be used for the current setting. The referenced setting must be declared in the metadata and is searched in ascending order. For details, refer to the 5.2.5. Service Settings section in the Specification;

- `type` - attribute defines setting value type. The `static-text` type dictates that value of the setting should not be changed by user and displayed as a static text. The `hidden` type dictates that neither setting name nor value can appear anywhere on application configuration screen. For details, refer to the 5.2.5.1 Data Types section in the Specification;
1. Group the settings basing on the `"/service/settings/group"` elements;
 2. Distinguish whether a setting is configurable only during installation or it can be reconfigured later (installation-only and regular settings), basing on the presence or lack of the `installation-only="true"` attribute;
 3. Distinguish whether the setting requires its value uniqueness within some scope. Such settings are marked with `uniq` attribute that value defines scope of setting value uniqueness. The Controller must validate and ensure such settings values uniqueness.
 4. If a verification script is declared in the `"/service/provision/"` section. The Controller may run it to get list of choices for `enum` setting type. The Controller should cache this list of choices and suggest it for respective setting editing. For details on verification script 5.3.3 Verification Script section of the Specification.
 5. Represent a setting in full compliance with how it is described in the metadata, basing *at least* on the values of the `"/setting/name"` element and the `"/setting[@type]"` attribute. For details, refer to the 5.2.5.1. Data Types section of the Specification.

When processing the settings data input by the user, the Controller should do the following:

- validate the submitted settings values according to the settings data types (the `"/setting[@type]"` attribute), and their optional restrictions (if any provided in the metadata, for example uniqueness).
If a verification script is declared, the Controller may run it itself to validate settings values or provide a user with dialog window where user can click special control for verification, for example, Verify button. A verification script definition may include `structured-output` element. In this case a script returns report on erroneous values in the form defined by the Specification. The Controller should catch output stream independently of error stream and process it. For details on script structured output, refer to the 5.3.2.3 Configuration script output section of the Specification;
- store the entered values of the settings in order to be able to present them during the instance reconfiguration;
- prepare environment variables `SETTINGS_<id>`, where `<id>` stands for a value of `"/setting[@id]"` attribute, set each variable to the corresponding input value, and pass to the application configuration script. If a setting has the `track-old-value` attribute Controller must pass to configuration script old value together with new one. Child services may reference to values of parent service settings using `value-of-setting` attribute. (For details, refer to the Specification's section 5.2.5. Service Settings.)

Processing Global Settings

The Controller must somehow store global settings of each application package available at its repository. (For details, refer to the Configuring Application's Global Settings (on page 19) section.)

When instantiating a package, the Controller must retrieve the package global configuration it stored, prepare from each global setting the `SETTINGS_<id>` environment variable, in just the same way it does for service settings (see above), and pass the variables to the configuration script. (For details, refer to the Specification's section 5.1.15. Global Settings.)

Provisioning Service

Method of service provisioning depends on hosting environment or application nature. Provisioning methods are declared with the `provision` element. The Specification defines the following methods: URL mapping and configuration script. These methods are not mutually exclusive and do not depend on each other. Thus, both of them can be declared for a service provisioning. URL mapping configures access to service through web and configuration script performs service installation.

If both methods are declared for a service provisioning, the order of methods invocation completely depends on Controller implementation.

Note: it is recommended, first install service to hosting environment and only then configure web access to it.

Application may declare different sets of provisioning methods depending on which requirements (set of requirements) were satisfied by Controller. This dependence of provisioning method on requirements is defined by `when-chosen` element that references to requirements branch with `requirements-id` attribute and declares appropriate provisioning methods. In other words, if the Controller has satisfied set of requirements with ID specified for `when-chosen` element, it should choose provisioning methods declared by this element. For details, refer to the 5.2.8 Service Provision section of the Specification.

Configuration Script

Configuration script is used by the Controller for service provisioning, if it is declared in the package metadata file `APP-META.xml` by the `"/service/provision/configuration-script"` element defined by the Specification in section 5.3.2. Configuration Script.

By the moment of script invocation, the following operations are to be performed by the Controller:

- Application files should be unpacked and deployed in hosting environment;
- All service requirements should be satisfied and requested resources allocated;
- Service settings are processed and transformed into respective environment variables. If URL mapping rules are declared for the service, respective environment variables for each mapping should be prepared as well.

Provisioning service operation with configuration script implies that the Controller performs the following steps:

- 1** *Determine configuration script interpreter.* To determine configuration script interpreter, the Controller reads from the package metadata the identifier of the script's language (the value of the `"/provision/configuration-script/script-language"` element). What interpreter must be used to run scripts written in a particular language is defined in the aspects, so the Controller must pass the identifier to the Aspects Parser. If the Aspects Parser supports the corresponding aspect, it returns the required interpreter information. Otherwise, the Controller must stop instantiating the application and roll back all the changes it has already done to the server.
- 2** *Run the script with the required interpreter passing to it all the required environment variables.* If the script interpreter is identified successfully, the Controller runs the configuration script with the `install` argument using the interpreter and passing to the script all the environment variables it prepared on the previous stages of instantiating. What variables must be passed to configuration script is defined by the Specification in section 5.3.2.2. Environment Variables.
- 3** *Check whether the script was executed successfully.* If the configuration script fails (returns non-zero exit code), it must be treated as a fatal error and the Controller must refuse to continue instantiating application. The Controller should also capture the script's `stdout` and `stderr` streams in order to log the error.

The configuration script definition may include `structured-output` element. In this case a script returns report on erroneous values in the form defined by the Specification. The Controller should catch output stream independently of error stream and process it. For details on script structured output, refer to the 5.3.2.3 Configuration script output section of the Specification.

The Controller should go to instantiating child service (if any) only after all parent services are instantiated in the hosting environment.

In case, URL mapping rules are declared for the service and web access to the service was not configured before configuration script execution, the Controller should go to performing service mapping. For details, refer to the URL Mapping section (on page 31) further in this guide.

URL Mapping

The Controller configures web access to the service basing completely on the rules of *URL Mapping* described in the package metadata file `APP-META.xml` by the `"/service/provision/url-mapping"` element defined by the Specification in section 5.3.1. URL Mapping. The Controller may perform this operation before or after executing service configuration script. The only restriction is that the Controller must prepare mapping environment variables and pass them to configuration script irrespective of whether the mapping has been performed or not.

Understanding the URL mapping on the level of package metadata format is what the Controller must at least do to properly copy the correct files on the site, and to properly configure web server. The Controller must select the package directories defined by the `"//mapping[@path]"` attributes, and, using an implementation-dependant technology, copy them to the hosting environment in such a way that the directories would be accessible at the corresponding URLs defined by each `"//mapping[@url]"`.

Each `"//mapping"` element (the root one, `"//url-mapping/mapping"`, and its children) in the package metadata can nest the 'URL handler' and 'permissions' elements defined by the aspects. To process the URL handlers and permissions, the Controller should use its functional part - Aspects Parser. More information about aspects is provided in the Controller Perspective section (on page 9). Under processing the URL handlers and permissions, we understand recognizing and performing actions necessary for implementing them, for example, setting specific access permissions on files, configuring web server, etc.

If the Aspects Parser comes over an unknown URL handler or permissions (i.e., the Aspects Parser has no info about such element and cannot recognize it) within at least one `"//mapping"` element, the Controller must abort instantiating the application.

To summarize it all, let us say that on the stage of provisioning service to hosting environment with URL mapping method, the following happens:

- 1 The Controller reads the URL mapping rules from the package metadata using the Package Metadata Parser and the Aspects Parser.
- 2 The Controller performs the mapping: properly copies the service files and processes permissions and URL handlers.

As soon as the Controller finishes with the mapping, it must prepare the `WEB_<id>_DIR` environment variables for each mapping to pass them to the configuration script then, as defined by the Specification in section 5.3.2.2.1. URL Mapping Variables. Also, the Controller must pass details of parent's URL mapping to configuration script of its child services.

The Controller must somehow store information on which files and folders of the application it copied to the hosting environment. This is required to distinguish the application files and the files created with this application, which will allow painlessly upgrading the application, replacing the old application files with new ones and leaving the created files untouched.

Possible Scenario

Conditions

The Control Panel has GUI.

The Controller is implemented so that it fully validates packages when they are added to Repository, including that it checks if a package uses additional aspects unknown to the Controller, and so on.

The Controller is implemented so that it stores all information on application package and its instances installed. It ensures access to application package for its instances while at least one instance exists.

The Control Panel is implemented so that the directories tree rooted in a directory devoted to the web site content represents the URL structure of the site as it is accessible from the Internet.

Starting Point. User initiates instantiating a package on a particular site.

Step 1. The Controller parses application metadata file `APP-META.xml`: extracts and organizes all information about the package. (Hereinafter, we will refer to this resulting information as to *application information*.)

Step 2. The Controller determines the service requirements and checks if the server and the site are capable of satisfying them.

Sub-step 2.1. Gets the information about service configuration script (`"/service/provision/configuration-script/script-language"`) and checks if there is an appropriate interpreter in the system and it can be used on the site to run the configuration script. If the package contains no configuration script, instantiating continues normally.

Sub-step 2.2. Gets the information about the service size when it is installed (`"/service/provision/url-mapping/installed-size/"`) and checks if the site has the amount of free disk space enough for installing the service.

Sub-step 2.3. Gets the list of service requirements (`"/service/requirements"`) and checks if the current site hosting conditions allow satisfying all the requirements from the list (for example, if the scripts support is enabled for the site, if the usage of such databases is allowed on the site, etc.). On this step, the Controller does not satisfy any of the requirements and does not allocate any resources, it just checks if all these actions are possible.

Here, the Controller creates a list of application requirements and hosting capabilities matching. It processes the requirement elements, comparing them with the hosting capabilities one after another. For each requirement, the Controller creates a kind of block with the `true|false` values defining if the requirement can be satisfied. Then the Controller puts the resulting blocks into the logical expression, basing on the requirements structure within `"/service/requirements"`, and computes it.

If the result is `false`, the Controller stops installing the application and returns to the user a list of requirements that cannot be satisfied.

If the result is `true`, the Controller creates a `CHOICE_<id>` variable for each requirements combination ("`choice/requirements`") which can be satisfied. If a requirements choice contains several satisfiable combinations, the Controller creates a variable for each, and groups them in order to give the user later a chance to choose which of these combinations to satisfy. For example, the application requires either MySQL or PostgreSQL database, and the hosting allows creating both. In such case, the Controller allows a user to choose what database he wants to be used by the application, see the **Sub-step 4.2**.

Step 3. The Controller checks if the service has an end-user license regulating the service usage ("`/service/license`"). If it has, the Controller creates a screen displaying the license text. If there is the `must-accept="true"` attribute in the application information, the screen contains the I agree to the license terms (...) checkbox and the submit and cancel buttons, and the Controller continues instantiating the package only if the user submits the selected checkbox.

Step 4. The Controller generates the screen with the installation configuration form.

Sub-step 4.1. Gets the list of all service settings within the "`/service/settings`" element. The Controller then creates a list of elements that the form will contain so that the user could enter the required settings values. For each setting data type the Controller makes the corresponding form element (text box, drop-down menu, etc.), adds to each element a label and tooltip from the required locale, group the elements in field sets.

Sub-step 4.2. Selects from the requirements matching list the groups of satisfiable requirement combinations (if any) and creates the form elements (a drop-down list for each group) that allow the user to choose which of these combinations to satisfy.

Sub-step 4.3. Selects from the list of satisfied requirements ones that require additional data, and makes the form elements with which the user will specify parameters of the resource to be allocated. (For example, if an application requires a database, the configuration form contains text fields for entering database administrator's login, password, password confirmations, etc.)

Sub-step 4.4. Makes the form elements with which the user will specify the application installation directory (site root relatively).

Sub-step 4.5. Collects the information on all the form elements made on the previous sub-steps and generates the screen.

Step 5. As the user enters the required information and submits the form, the Controller gets the submitted application data and validates it.

Sub-step 5.1. Validates the block of requirements-related data. For example, if creating a database, the Controller checks if a databases with the name same as the entered one does not exist. If any of such checks fails, the Controller adds a message about it to the error stack.

Sub-step 5.2. Checks if the installation directory specified by the user does not exist. If the directory exists, the Controller adds a message about it to the error stack.

Sub-step 5.3. Validates each application setting value basing on the settings information from the package metadata. If an invalid value is set, the Controller adds a message about it to the error stack.

Sub-step 5.4. Checks the state of the error stack. If it contains any messages, the Controller returns to the **Step 4**: generates the installation form again adding to it the error stack's content. If the error stack is empty, the Controller continues to the next step.

Step 6. The Controller further processes the installation data entered by the user.

Sub-step 6.1. Satisfies the service requirements: enables scripting support on the site, allocates all the required resources, and stores the information about them (for the purpose of future deallocating or re-allocating).

Sub-step 6.2. Prepares all the necessary environment variables.

Step 7. The Controller deploys the application files to the site location defined by user, following the instructions defined in the `"/service/provision/url-mapping"`. The Controller properly copies the files and performs the actions required to set URL handlers and permissions. While deploying the application, the Controller prepares the required mapping-related environment variables, and records the information on each directory where the application files are copied. When finished deployment, the Controller saves this information for the purpose of future application upgrade or removal.

Note: application web access is configured, but not activated.

Step 8. The Controller gets the application global configuration stored in the Repository, analyzes it and prepares global-configuration environment variables.

Step 9. The Controller invokes the service configuration script to configure the service instance.

Sub-step 9.1. Gathers all the information about environment variables prepared on the previous steps, and creates the variables basing on the rules defined in the Specification.

Sub-step 9.2. With the interpreter determined on the **Sub-step 2.1.**, the Controller runs the configuration script in the environment where all the environment variables created on the previous sub-step are defined.

Sub-step 9.3. If the script invocation succeeds, the Controller displays to the user the screen with the successful instantiation message and congratulations.

Step 10. The Controller behavior on this step depends on whether the previous step was completed successfully.

Step 10.1. If the previous step was completed successfully, the Controller performs the following post-installation operations:

- copies the application metadata file and the scripts directory from the application package to the site-specific location, in order to make the instance operable regardless of the package state;
- selects from the local application configuration all regular settings and stores them to the site-specific location;
- activates application web access.

Step 10.2. If the previous step was completed with an error, the Controller rolls back all the changes it made to the system and the site, displays the error message to the user, and returns to **Step 4**. Rolling back the changes implies removing all application files copied to the site, and releasing all allocated resources.

Removing Application Instances

Removing application instance implies removing application 'root' service. When removing the service instance, the Controller must invoke respective configuration script with the `remove` argument before it removes the application files and deallocates resources used by it, as defined by the Specification in section 5.3.2.1.4 Canceling Service. The Controller must pass to the configuration script all application information transformed into respective environment variables: settings (`SETTINGS_<id>`), except marked as installation-only; application URL mapping environment variables (if any), as defined by the Specification in section 5.3.2.2. Environment Variables. After configuration script successful execution, the Controller must remove application files and free allocated resources.

If an application package contains separate scripts for application services, it is possible to remove certain service. When user initiates selected service removal, the Controller must check first whether the service has child ones. If child services are present the Controller may remove them or notify the user that service can not be removed due to child services existence and offer their removing. Child services removal procedure completely depends on Controller implementation. To remove a service the Controller must invoke configuration script specified for selected service in the package metadata with the `remove` argument and pass to it all needed environment variables. After configuration script successful execution, the Controller must remove service URL mapping, service files and free allocated resources.

For details on configuration script invocation, refer to the Configuration Script section (on page 30) earlier in this guide.

Possible Scenario

Conditions.

The Controller is implemented so that it stores all the data used and created during the application instance creating, reconfiguring, and upgrading, excluding the data related to the installation-only settings. Also, the Controller stores the application metadata and scripts at the instance-specific locations.

The application has only one service.

Starting Point. User initiates removing application instance from a particular hosting environment.

Step 1. The Controller disables access to application instance and removes application instance URL mapping.

Step 2. The Controller executes service configuration script.

Sub-step 2.1. From the application metadata stored on the site, the Controller gets information about configuration script language ("`/application/service/provision/configuration-script/script-language`"), and checks if there is an appropriate interpreter in the system and if it can be used on the environment to run the configuration script.

Sub-step 2.2. Gets all the application data created during the instance provisioning, upgrades and changes, such as the settings data (excluding installation-only settings), data about resources allocated to the instance and other changes made by the Control Panel to the system (web server configuration, permissions, etc.). Basing on this data, the Controller creates environment variables as defined by the Specification.

Sub-step 2.3. Using the interpreter, the Controller runs the configuration script with the `remove` argument, and passes to it all the environment variables it created on the previous sub-step.

Sub-step 2.4. If the script invocation fails, the Controller returns an error, stops removing application and displays to the user the error message. Otherwise, it proceeds to the next step.

Step 3. The Controller removes the application instance files and folders.

Sub-step 3.1. Gets the information on the application files and folders copied from the application package to the site during the installation. Basing on this information, the Controller creates two file trees, one representing the files and folders of application itself, another the files and folders created with the application instance.

Sub-step 3.2. In the form of a dialogue screen, the Controller asks the user whether he wants to remove all files related to the application instance, or remove only application files and keep the files created by the application instance.

Sub-step 3.3. Depending on the user's choice, the Controller removes either all instance-related files and folders, or only those of the application itself.

Step 4. The Controller rolls back the changes made to the system with the purpose of the instance proper functioning.

Sub-step 4.1. Retrieves information about the resources allocated to the application instance, and frees them.

Sub-step 4.2. Retrieves information about other changes made to the system (e.g., web server configuration), and rolls them back.

Updating Application Instances

The Controller should implement a mechanism of updating an application instance to the newer version in case that a suitable update package is available. APS supports two types of application updates: *patch* and *upgrade*. Generally speaking, the difference is that patching supposes unattended instance update, while upgrading brings more serious changes to the instance and may require the user attendance. Application patch and upgrade are defined in detail by the Specification in section 5.1.15. Updates. Distinguishing patches and upgrades was made with the purpose to facilitate implementation of the Controller update function. Depending on what kind of update is required - patch or upgrade - the Controller behavior slightly differs.

Updating an application instance supposes that the Controller performs the following two steps:

- 1 *Resolving update packages*. On this step, the Controller understands that a particular package, or packages, can be used for patching/upgrading the instance.
- 2 *Updating the application instance*. This is the very process of update, which supposes that the application of older version is replaced with the one of newer version, and the user settings and files of the old application are picked up by the new one. The Controller's behavior on this step depends on whether the application instance is patched or upgraded and whether a particular update mode is required.

Further in the chapter, we describe what's specific in the Controller implementation about *patching* and *upgrading* instances. In case when some information concerns both, we use the terms *update* and *updating*.

Resolving Update Packages

The Controller should understand whether a package is suitable for updating a particular application instance or not.

The first thing that the Controller should do here is to make sure it is dealing with the instance and the package of the same application. This is done by comparing their unique identifiers and the packager's URIs defined by the `"/application/id"` and `"/application/packager/uri"` elements in the package metadata. Because the `"/application/packager/uri"` element is optional and can be missing, collisions may appear, which the Controller must resolve basing on the following rules:

- if the instance packager's URI matches the one of the package, it's the same application;
- if both the instance and the package lack packager's URIs, treat them as ones of the same application;
- if the instance lacks packager's URI and the package has one, treat them as ones of the same application;
- if the instance has packager's URI and the package lacks it, treat them as ones of the different applications;

As soon as the instance and the package are identified as the same application, the Controller should understand whether the package is suitable for patching or upgrading the instance. This understanding is gained basing on the `match` attribute value of `"/upgrade"` and `"/patch"` elements in the package metadata file. The `match` attribute value is a logical expression that comprises application version, release and logical operators (`=`, `<`, `>`, and `or`). The Controller should parse this expression and define either exact version and release of the application or the minimum version and release of the application to be updated.

To summarize it all, let us say that the Controller on the stage of defining whether the package is suitable for updating the instance, the following happens:

- The Controller reads package metadata file to find `"/upgrade"` and `"/patch"` elements. If none of these elements is present then the package cannot be used in any updates;
- The Controller finds the `"/patch"` element. It parses the `match` attribute value and extracts application version and release. The next step is to compare application instance version and release with extracted ones, if values matches, the application instance can be patched.
- The Controller finds the `"/upgrade"` element. It parses the `match` attribute value and extracts application version and release. The next step is to compare application instance version and release with extracted ones, if values matches, the application instance can be upgraded.

The elements are defined by the Specification in section 5.1.15. Updates.

Patching and Upgrading an Instance

Starting APS 1.1 updating application instance comprises updating its services instances. Necessity of updating particular service instance depends on provisioning method specified for it, as follows:

- If service declares the URL Mapping as provisioning method in package metadata file, the service instance requires individual update procedure;
- If service declares any other provisioning methods in package metadata file, the service instance is updated automatically during update of its parent service, meaning that the Controller must not perform any update procedures for this service.

To make describing the application update easier, we will use words *old* and *new* to designate anything related to the application already installed and the package to which it is updated, correspondingly. For example, if talking about service settings, *old settings* means the collection of settings that belong to the provisioned service instance, and *new settings* means those defined in the update package.

The following are the important points of the Controller's Update function implementation:

Resolving settings.

When updating an application instance, the Controller must properly resolve the application installation settings, having in view the following:

- The Controller should strictly follow the settings descriptions in package metadata (defined by the Specification in section 5.2.5. Service Settings), no matter whether the settings are old or new. First of all, this concerns the old installation-only settings, which are still supposed not to be changed, even during application upgrade.
- In order to avoid collisions with configuration during update, the Specification prohibits:
 - changing the same service settings from installation-only to regular, and vice versa, from package to package. This means that if an old service has an installation-only setting identified by the ID, for example, `main-title` (`<setting id="main-title" installation-only="true"> ... </setting>`), the following versions will have exactly the same installation-only setting with the same ID (or no setting with such ID at all).
 - replacing setting value with reference to value of another setting. This means that if an old service settings were lack of the `value-of-setting` attribute, the following versions must not add this attribute to these service settings.
- To support the unattended application update during the patching, the Specification prohibits appearing new settings without default values in the patch packages. This means that when patching an instance, the Controller must automatically pass the default values of the new settings to the configuration script.

Resolving requirements.

The Controller must correctly resolve the requirements of the update package as follows:

- When patching, the Controller must skip reading new requirements, and preserve all resources allocated to the old version so that the new version could painlessly pick them all up, as defined by the Specification in section 5.2.7. Requirements.
- When upgrading, the Controller must carefully analyze new requirements and provide a mechanism of satisfying them together with migrating the application vitally-important data. For example, when a new application version requires using a database of another version, the Controller must ensure that both databases are accessible during upgrade.

For general information on application requirements, refer to the Determining and Satisfying Requirements section (on page 25). What's specific about resolving requirements during upgrade is that the Controller must preserve the old allocated resources so that the new application could use them. How should a resource of a particular type be preserved is defined in the aspect devoted to that type.

Removing old application files and deploying the new ones.

Starting APS 1.2 an update declaration may include `mode` attribute. The value of this attribute defines the update procedure:

- *simple* - the `mode` attribute is absent. Deploying files from a new package during update is quite similar to provisioning service: the Controller must read the URL mapping from the package metadata and perform all the required actions (for details, refer to the URL Mapping section (on page 31)). The difference is that the Controller must properly replace the old files with the new ones. For details, refer to the section 5.3.1.2. Update of the Specification.
- *managed* - both old and new application directories are available during configuration script execution. The only restriction is old and new application directories should reside on the same file system. The old application directories are removed after configuration script successful execution. Paths to old and new application installations are passed to the script in `OLDWEB_<id>_DIR` and `WEB_<id>_DIR` environment variables, respectively.
- *backup* - the update procedure is a bit complicated then the *managed* one. It involves backing up old application data, new instance creation, then restoration of backed up data on new instance and configuration script execution. Paths to old and new application installations are passed to the script in `OLDWEB_<id>_DIR` and `WEB_<id>_DIR` environment variables, respectively.

Consider adding to Controller implementation possibility to organize FIFO technology usage for updating an application instance in `backup` mode. This will greatly speed up updating procedure.

Running the service configuration script.

The finishing step the Controller must perform to complete an application update is invoking the new configuration script, as defined by the Specification in section 5.3.2.1.2. Upgrading Application. Using the appropriate interpreter, the Controller must run the configuration script with arguments `"upgrade <old version> <old release>"`, where `<old version>` and `<old release>` stand for the version and release of application from which the upgrade is performed, correspondingly. For details on configuration script invocations, refer to the Configuration Script section (on page 30).

Possible Scenario

Conditions.

The Controller is implemented so that it provides the Application Update user interface, which allows a user to ask for the list of available updates for a particular application instance, select a required patch or upgrade from the list and initiate the instance update.

The Controller is implemented so that it stores all information about old package and the application instance:

- the application metadata;
- information about all application files deployed on the site;
- information about all old requirements satisfactions;
- information about all services installation-only settings, separately from the application files and folders, so that when the application files are removed, the values of old installation-settings are still available.

The Control Panel is implemented so that the directories tree rooted in the web site home directory represents the URL structure of the hosting environment as it is accessible from the Internet.

Starting Point. User initiates opening a list of updates available for a particular application instance.

Step 1. The Controller resolves update packages for the instance and displays a list of available updates:

Sub-step 1.1. Searches through metadata of all application packages in Repository to find those of the same application as the application instance but different version and release (basing on the values of `"/application/name"`, `"/application/packager/uri"`, `"/application/version"`, `"/application/release"` elements in metadata files).

Sub-step 1.2. From the packages found on the previous sub-step, the Controller selects the patch and upgrade packages:

- adds to the list of patches each package that lists the instance version and release as patchable or indicates the minimum version and release from which patching is possible equal or lower to the instance ones (match attributes of any `"/patch"` element);
- adds to the list of upgrades each package that lists the instance version and release as upgradable or indicates the minimum version and release from which upgrading is possible equal or lower to the instance ones (match attributes of the `"/upgrade"` element).

Sub-step 1.3. Displays the list of available updates to the user, grouping the available packages as patches and upgrades. The list contains controls which allow user to select a package to which the instance must be updated and initiate the process of application update. Depending on whether a patch or upgrade is selected, the Controller starts the process of patching or upgrading.

The following scenario steps that describe patching are marked with the *P* letter, those that describe upgrading, with the *U* letter, those that are performed in both cases are just numbered and don't have any letter marks.

Step 2. The Controller backs up old instance data.

Step 3. The Controller parses application metadata file `APP-META.xml` of the update package: extracts and organizes all information about the package.

Step 4-U. The Controller determines the new requirements and checks if they are satisfied with the current conditions.

Sub-step 4-U.1. Gets information about service configuration script (`"/service/provision/configuration-script/script-language"`) and checks whether there is an appropriate interpreter in the system and it can be used on the hosting environment to run the configuration script. If the check fails, the Controller adds a message about it to the error stack.

Sub-step 4-U.2. Gets information on the disk space occupied by the new application when it's installed (`"/service/provision/url-mapping/installed-size"`) and checks if there is enough free disk space in the hosting environment: compares whether the new installation size is smaller than the sum of the site free disk space and the old installation size. If the check fails, the Controller adds a message about it to the error stack.

Sub-step 4-U.3. Gets the list of new requirements (`"/service/requirements"`) and compares them to the old ones in order to check whether the old requirements satisfactions satisfy them. If there are any new requirements that need satisfaction, goes to the **Sub-step 4-U.4**; if no, goes to the **Sub-step 4-U.5**.

Sub-step 4-U.4. Checks if the current hosting environment conditions allow satisfying the new requirements (for example, if the scripts support is enabled for the site, if the usage of such databases is allowed on the site, etc.).

If the check succeeds, the Controller satisfies the requirements and performs all the necessary additional operations (e.g., migrates the database from the old database server to the new one). Then stores all the information about them (for the purpose of future deallocating or re-allocating) and prepares all the necessary environment variables which concern the satisfied requirements.

If the check fails, the Controller adds a message about it to the error stack.

Sub-step 4-U.5. Checks the error stack status.

If it contains any messages, the Controller stops the process of upgrading the application and displays the screen with the error stack content.

If there are no messages, the Controller continues to the next step.

Step 5. The Controller checks whether the new service has an end-user license which regulates the service usage (`"/service/license"`). If it has, the Controller creates a screen displaying the license text. If there is the `must-accept="true"` attribute, the screen contains the I agree to the license terms (...) checkbox and the submit and cancel buttons, and the Controller continues updating only if the user submits the selected checkbox.

Step 6-P. The Controller checks if the new package metadata contains any installation-only settings which differ (according to the setting ID) from the old installation-only settings (the `//setting` elements that have the `installation-only="true"` attribute). If it does, the Controller prepares for such settings environment variables with default values taken from the package metadata. If no default values found, the Controller skips the step.

Step 6-U. The Controller checks if the instance upgrade requires configuring additional settings, and if so, presents the user with configuration form and processes the data submitted by user.

Sub-step 6-U.1. From the upgrade package information, gets the list of all service installation-only settings (the `//setting` elements that have the `installation-only="true"` attribute).

Sub-step 6-U.2. Retrieves the stored list of old installation-only settings, compares it to the list created on the previous sub-step, and selects the new installation settings contained in the new package (the ones the old package didn't have) if there are any.

Sub-step 6-U.3. For each new installation setting, creates a list of elements that the form will contain so that the user could enter the required settings values. For each setting data type the Controller makes the corresponding form element (text box, drop-down menu, etc.), adds to each element a label and tooltip from the required locale, and so on. And generates the form.

Sub-step 6-U.4. As the user enters the required information and submits the form, the Controller gets the submitted data and validates each setting value basing on the settings information from the package metadata. If an invalid value is set, the Controller returns to the previous sub-step: Generates the installation form again adding to it messages about wrong values.

Sub-step 6-U.5. From the upgrade package information, gets the list of all service settings that lack the `installation-only` attribute.

Sub-step 6-U.6. Retrieves the stored list of old settings, compares it to the list created on the previous sub-step, and selects the new settings contained in the new package (the ones the old package didn't have) if there are any.

Sub-step 6-U.7. For each new setting, creates a list of elements that the form will contain so that the user could enter the required settings values. For each setting data type the Controller makes the corresponding form element (text box, drop-down menu, etc.), adds to each element a label and tooltip from the required locale, and so on. And generates the form.

Sub-step 6-U.8. As the user enters the required information and submits the form, the Controller gets the submitted data and validates each setting value basing on the settings information from the package metadata. If an invalid value is set, the Controller returns to the previous sub-step: Generates the installation form again adding to it messages about wrong values.

Sub-step 6-U.9. Prepares all the necessary environment variables concerning the settings.

Step 7. The Controller deploys the new application files to the site.

Sub-step 7.1. Gets the stored information about old application files, and removes them.

Sub-step 7.2. Deploys the new application files following the instructions defined in the update package `"/provision/url-mapping"` rules. Properly copies the files and performs the actions required to set URL handlers and permissions. While deploying the application, the Controller prepares the required mapping-related environment variables, and records the information on each directory where the application files are copied. When deployment is finished, the Controller saves this information for the purpose of future application upgrade or removal.

Note: application web access is configured, but not activated.

Step 8. The Controller gets the new application global configuration stored in the Repository, analyzes it and prepares global-configuration environment variables.

Step 9. The Controller invokes the application configuration script to configure the application.

Sub-step 9.1. Gathers all the information about environment variables prepared on the previous steps and other required variables (full instance URL, for example), and creates the variables basing on the rules defined in the Specification.

Sub-step 9.2. Runs the configuration script in the environment where all the environment variables created on the previous sub-step are defined, passing to it the old application version and release as arguments.

Sub-step 9.3. If the script invocation succeeds, the Controller activates application web access and displays to the user the screen with the successful update message and congratulations. If the script invocation fails, the Controller restores old instance from the back up and displays the screen with the error message.

Step 10. The Controller organizes access to the new metadata file and the scripts directory from the update package for the instance.

Configuring Application Instance

Application instances configuring is initiated in the following two cases:

- 1 Automatically, when the application package global configuration is changed, the Controller must initiate reconfiguring all the instances created from this package according to the changes. For details on changing global settings which leads to the very reconfiguring of instances, refer to the section *Configuring Application Global Settings* (on page 19).
- 2 Manually, when a user wants to change particular service settings.

In such case, the Controller provides UI for user to enter the values of the service settings (described in metadata with `//service/settings/setting` elements which lack `installation-only="true"` attribute), validates the entered values, ensures settings values uniqueness (if required), and initiates the reconfiguring. The Controller must store the new settings values entered by user in order to present them on demand (in most cases, during the following application reconfiguration).

In both cases, the Controller performs just the same actions: it applies new values of settings to an application instance (or several instances). The only differences are the actions leading to the reconfiguring and the type (global or service) of settings applied.

As soon as reconfiguring application instance is initiated and the Controller has new settings values, validated, it must prepare for each setting an environment variable `SETTINGS_<id>` where `<id>` replaces the value of the setting's `id` attribute (`//setting[@id]`). Then the Controller must run the service configuration script with the `configure` argument and pass to it all the required environment variables, as defined by the Specification in the 5.3.2. Configuration Script and 5.3.2.1.3 Changing Settings sections.

Possible Scenario

Conditions.

Control panel is implemented so that it allows a user to open a screen with the form of a particular application service instance (installed on a particular hosting environment) configuration, where the user can specify new service settings.

Controller is implemented so that it stores all the instance-related information in its internal database, including the values of service settings (defined during instantiating or the following reconfiguring).

Starting Point. User initiates opening the service configuration form.

Step 1. The Controller gathers all information necessary for generating the form:

Sub-step 1.1. Selects from the database the settings of the target service instance.

Sub-step 1.2. Processes the settings info (setting name, description, data type, grouping information, value), and generates the form: all settings in the form of particular form elements (drop-down lists, check boxes, etc.) with the old setting values filled in.

Step 2. The Controller passes the form code to CP, which processes it and presents the result to the user as the Service Configuration screen.

Step 3. As soon as the user fills in the form and submits it, the Controller validates the submitted data:

Sub-step 3.1. Validates each setting value basing on the settings type information and for those settings that have `uniq` attribute ensures their values uniqueness. If an invalid value is set, the Controller adds a message about it to the error stack.

Sub-step 3.2. Checks the error stack status: If it's not empty, the Controller returns to **Step 2** adding to the form the content of the error stack.

Step 4. The Controller prepares all the environment variables required by the service configuration script:

Sub-step 4.1. Prepares the settings variables `SETTINGS_<id>` setting them to the entered settings values.

Sub-step 4.2. Reads the other instance-related information from the database and if any are found, prepares the rest of the variables (describing the full instance URL, and so on).

Step 5. The Controller invokes the service configuration script to configure the service instance with the new setting values.

Sub-step 5.1. Gathers all the information about environment variables prepared on the previous step, and prepares the variables basing on the rules defined by the Specification.

Sub-step 5.2. Runs the configuration script in the environment where all the environment variables prepared on the previous sub-step are defined.

Sub-step 5.3. If the script invocation succeeds, the Controller displays to the user the screen with the successful reconfiguration message and congratulations.

Step 6. The Controller stores the new settings in its database.

Enabling and Disabling Service

Configuration script declared for particular service may be able to change this service status. This configuration script ability is declared with the `status-control` element in the package metadata file. For details, refer to the 5.3.2.1.5 Enabling/Disabling Service section in the Specification.

When the Controller determines that the service status may be changed, the use and storage of this information completely depends on the Controller implementation. For example, it may provide to end-user some control for either service disabling or service enabling or both operations. The other use case, the Controller changes service status basing on end-user account balance: disables service, if user is lack of funds and enables it, when its balance is replenished.

The Specification provides the only requirement to disabled service that it is must not serve to its users or operate on behalf of service owner. If the service uses some hosting resources, whether to leave them allocated or to free, depends on the Controller implementation.

The procedure of changing service status just slightly differs from other configuration script invocations. Depending on service current status, the Controller invokes configuration script with the `enable` or `disable` argument. The Controller must pass to the configuration script all service information transformed into respective environment variables: settings (`SETTINGS_<id>`), except marked as installation-only; application URL mapping environment variables (if any), as defined by the Specification in section 5.3.2.2. Environment Variables.

For details on configuration script invocations, refer to the Configuration Script section (on page 30).

Possible Scenario

Conditions. The Controller is implemented so that it stores all the data used and created during the application instance creating, reconfiguring, and upgrading, excluding the data related to the installation-only settings. Also, the Controller stores the application metadata and scripts at the instance-specific locations.

Starting Point. Disabling service instance is initiated.

Step 1. The Controller executes respective configuration script.

Sub-step 1.1. From the application metadata stored on the site, the Controller gets information about configuration script language (`"/application/service/provision/configuration-script/script-language"`), and checks if there is an appropriate interpreter in the system and if it can be used on the environment to run the configuration script.

Sub-step 1.2. Gets all the application data created during the instance creation, upgrades and changes, such as the settings data (excluding installation-only settings), data about resources allocated to the instance and other changes made by the Control Panel to the system (web server configuration, permissions, etc.). Basing on this data, the Controller creates environment variables as defined by the Specification.

Sub-step 1.3. Using the interpreter, the Controller runs the configuration script with the `disable` argument, and passes to it all the environment variables it created on the previous sub-step.

Sub-step 1.4. If the script invocation fails, the Controller returns an error, stops disabling service and displays to the user the error message.

Application Data Retention

Application packager may include a special script into the package that serves for backing up application user data and further its restoration on new application instance in case of some failures, updates or migration. These operations are performed with the help of backup script on application 'root' service instance. Migration operation is a sequential execution of backing up and restoring operations in this regard.

The backup script is declared with the `"/service/provision/backup-script"` element defined by the Specification in section 5.3.5. Backup/Restore Script. Its declaration must include programming language it is written in. The script is invoked on instance backing up or restoring.

The details of Controller behavior specific for invoking backup script during a particular operation are considered further in this section.

Backing Up Service Instance

On executing backup operation the script should back up all sensitive application data to a file. The Controller should ensure that the file is writable. The path to this file is provided in the `BACKUP_FILE` environment variable. How this variable gets its value (manually by user or automatically by a Controller) is completely on the Controller implementation. Same set of variables as for configuration script should be passed to the backup script on its invocation.

Application resources like databases or mailboxes should be backed up as well. If the backup script declaration includes the `sufficient` element, such resources are serialized into backup file by means of the script. If the element is absent, it is the Controller responsibility to back up all involved resources. Consider this possibility in the Controller implementation.

The Controller should assure that application instance exists before starting backup operation.

Backing up service operation with backup script implies that the Controller performs the following steps:

- 1** *Determine backup script interpreter.* To determine backup script interpreter, the Controller reads from the package metadata the identifier of the script's language (the value of the `"/provision/backup-script/script-language"` element). What interpreter must be used to run scripts written in a particular language is defined in the aspects, so the Controller must pass the identifier to the Aspects Parser. If the Aspects Parser supports the corresponding aspect, it returns the required interpreter information. Otherwise, the Controller must stop backing up service and roll back all the changes it has already done to the server.
- 2** *Run the script with the required interpreter passing to it all the required environment variables.* If the script interpreter is identified successfully, the Controller runs the backup script with the `backup` argument using the interpreter and passing to the script all the environment variables. What variables must be passed to backup script is defined by the Specification in the 5.3.5.2. Environment Variables section.

- 3 *Check whether the script was executed successfully.* If the backup script fails (returns non-zero exit code), it must be treated as a fatal error and the Controller must refuse to continue operation. The Controller should also capture the script `stdout` and `stderr` streams in order to log the error.

The configuration script definition may include `structured-output` element. In this case a script returns report on erroneous values in the form defined by the Specification. The Controller should catch output stream independently of error stream and process it. For details on script structured output, refer to the 5.3.2.3 Configuration script output section of the Specification.

Restoring Service Instance

Restoring of previously backed up data performed also with the backup script. Operation prerequisites and the Controller actions are much the same as for backing up operation. The only difference is the backup script is invoked with `restore` argument and actually restores data from the file specified in the `BACKUP_FILE` environment variable. The Controller should ensure that the file is readable.

If the backup script declaration includes the `sufficient` element, resources like mailboxes and databases are created empty by the Controller, but filled with backed up data by means of the script. If the element is absent, it is the Controller responsibility to restore all involved resources. Consider this possibility in the Controller implementation.

Migrating Service Instance

A service instance migration is a sequential execution of backing up and restoring instance on another hosting environment. Consider adding to Controller implementation possibility to organize FIFO technology usage for instance migration. This will greatly speed up operation execution.

Possible Scenario

Conditions.

The Control Panel is implemented so that it initiates backup procedure for certain application instance periodically and provides the Controller with the path to backup file.

The Controller is implemented so that it stores all information about package and the application instance:

- the application metadata;
- information about all application files deployed on the site;
- information about all requirements satisfactions;
- information about all services installation-only settings, separately from the application files and folders, so that when the application files are removed, the values of old installation-settings are still available.

The Control Panel is implemented so that the directories tree rooted in the web site home directory represents the URL structure of the hosting environment as it is accessible from the Internet.

Starting Point. Control Panel initiates backup procedure for a particular application instance and points the Controller the location of backup file.

Step 1. The Controller parses application metadata file `APP-META.xml`.

Sub-step 1.1 The Controller gets information about service backup script ("`//service/provision/backup-script/`") and checks whether the `sufficient` element is provided. If the element is present the Controller proceeds to the next step. Otherwise, it backs up application instance resources by its means. If backing up of resources fails, the Controller stops operation and returns error message.

Sub-step 1.2 The Controller gets information about service backup script language ("`//service/provision/backup-script/script-language`") and checks whether there is an appropriate interpreter in the system and it can be used on the hosting environment to run the script. If the check fails, the Controller refuses to continue the operation.

Step 2. Gathers all the information about environment variables stored (the same set as for configuration script, full instance URL, for example), and creates the variables basing on the rules defined in the Specification.

Step 3. The Controller invokes the application backup script passing to it all required environment variables.

Sub-step 3.1. The Controller checks that backup destination file is writable.

Sub-step 3.2. Runs the backup script in the environment where all the environment variables created.

Sub-step 3.3. If the script invocation succeeds, the Controller returns to Control Panel successful message. If the script invocation fails, the Controller returns error message and deletes backup destination file. If the Controller was responsible for resources back up it is deleted as well.

Getting Resources Usage Report

Monitoring usage of hosting resources is performed by platform itself, but hosting platform has no knowledge of an application internal resources. For example, Open-Xchange application has *InfoStore* module. This module allows application users to place some items for storage and share them with others. A storage comprises a number of folders each of them occupies some storage space. So, storage space is an application resource that usage can be reported by application. The other example of such resource can be a number of application users or mailboxes. A hosting provider may want to additionally charge a user for such resources usage.

The `resources` section of package metadata file may describe resources that usage is to be reported by application service instance. Resource definition may include unit of measure ("`//resource[@class]`" attribute) and setting that limits resource usage ("`//resource[@limiting-setting]`" attribute). For details, refer to the 5.2.6 Resources section of the Specification.

The information on resources usage is collected by resource script that is called by Controller. Application may provide the Controller with a hint how often it should run the script. This period is defined with the `poll-interval` attribute of the script.

The following issues are at discretion of implementer:

- how the list of resources and statistics of their usage are to be stored;
- how procedure of collecting usage statistics is initiated;
- how the Controller reports collected statistics to a hosting platform.

Getting resource usage operation with resource script implies that the Controller performs the following steps:

- 1** *Determine resource script interpreter.* To determine resource script interpreter, the Controller reads from the package metadata the identifier of the script language (the value of the "`//provision/resource-script/script-language`" element). What interpreter must be used to run script written in a particular language is defined in the aspects, so the Controller must pass the identifier to the Aspects Parser. If the Aspects Parser supports the corresponding aspect, it returns the required interpreter information. Otherwise, the Controller must stop instantiating the application and roll back all the changes it has already done to the server.
- 2** *Run the script with the required interpreter.* If the script interpreter is identified successfully, the Controller runs the resource script using the interpreter.
- 3** *Process the script output.* The script returns report on resources usage that matches XML schema defined in the Specification. The Controller gets the output and processes it. The further script output processing is completely depends on a Controller implementation.

For details on resource script invocation and output structure, refer to the 5.3.4 Resource Script section of the Specification.

Possible Scenario

Conditions.

The Control Panel is implemented so that it initiates the procedure of collecting applications instances resources usage.

The Controller is implemented so that it stores the list of resources linked to corresponding instances along with resources usage statistics.

The Control Panel is implemented so that the directories tree rooted in the web site home directory represents the URL structure of the hosting environment as it is accessible from the Internet.

Starting Point. Control Panel requests the Controller to get current usage of a particular resource.

Step 1. The Controller checks its resources storage whether requested resource is present. If the resource is absent, the Controller refuses to continue the operation and returns respective error message. If resource is present the Controller determines instance it belongs to.

Step 2. The Controller parses respective application metadata file `APP-META.xml` to get the information about the resource script ("`//service/provision/resource-script/script-language`") and checks if there is an appropriate interpreter in the system and it can be used on the hosting environment to run the script.

Step 3. The Controller invokes the application resource script to get the report.

Sub-step 3.1. Runs the resource script in the environment with respective interpreter.

Sub-step 3.2. If the script invocation succeeds, the Controller checks whether script output can be parsed. If the script invocation fails or script output is not parsed, the Controller returns to Control Panel error message.

Step 4. The Controller parses script output to extract requested resource current usage, updates its storage with new value and passes it to Control Panel.

APPENDIX A

Appendix. Sample APP-META.xml

For the purpose of illustrating what environment variables the Controller must to create and pass to configuration script on its invocation, we present the file with metadata of the SugarCRM application. This application provides 'root' service named *instance* that in its turn provides child service named *account*. For each of services declared a separate configuration scripts are specified: `configure` for 'root' service and `usermanager` for child one. Depending on what script is invoked the Controller must prepare the following sets of environment variables:

'Root' service variables:

- Pre-defined variables representing the instance: URL:
BASE_URL_SCHEME
BASE_URL_HOST
BASE_URL_PORT
BASE_URL_PATH
- Variables defined by service settings declarations:
SETTINGS_admin_name
SETTINGS_admin_password
SETTINGS_title
SETTINGS_send_usage_statistics
SETTINGS_check_for_updates
- Variables defined by service requirements declarations:
PHP_VERSION
DB_main_NAME
DB_main_LOGIN
DB_main_PASSWORD
DB_main_HOST
DB_main_PORT
DB_main_VERSION
DB_main_PREFIX
- Variables defined by service URL mapping declarations:
WEB__DIR
WEB__cache_DIR
WEB__custom_DIR
WEB__data_DIR
WEB__modules_DIR
WEB__tmp_DIR
WEB__config.php_DIR

Child service variables:

- Pre-defined variables representing the instance URL:
BASE_URL_SCHEME
BASE_URL_HOST
BASE_URL_PORT
BASE_URL_PATH
- Variables defined by service settings declarations:
SETTINGS_user_login
OLDSETTINGS_user_login
SETTINGS_user_password
SETTINGS_first_name
SETTINGS_last_name
SETTINGS_user_email
SETTINGS_title
SETTINGS_department
SETTINGS_phone_work
SETTINGS_phone_mobile
SETTINGS_phone_fax
SETTINGS_phone_home
SETTINGS_address_street
SETTINGS_address_city
SETTINGS_address_state
SETTINGS_address_country
SETTINGS_address_postalcode
SETTINGS_description
- Variables defined by parent service requirements declarations:
PHP_VERSION
DB_main_NAME
DB_main_LOGIN
DB_main_PASSWORD
DB_main_HOST
DB_main_PORT
DB_main_VERSION
DB_main_PREFIX

- Variables defined by parent service URL mapping declarations:

```

WEB___DIR
WEB__cache_DIR
WEB__custom_DIR
WEB__data_DIR
WEB__modules_DIR
WEB__tmp_DIR
WEB__config.php_DIR

```

The following is the full content of the SugarCRM's APP-META.xml file where each variable-producing place is provided with a comment saying what environment must be created there and why.

```

<application xmlns="http://apstandard.com/ns/1" version="1.2">
  <!-- Full URL specifying where the application is to be installed
  -->
  <!--
  BASE_URL_SCHEME
  BASE_URL_HOST
  BASE_URL_PORT
  BASE_URL_PATH
  -->
  <id>http://www.sugarcrm.com/crm/</id>
  <name>SugarCRM</name>
  <version>5.2.0a</version>
  <release>1</release>
  <homepage>http://www.sugarcrm.com/crm/</homepage>
  <vendor>
    <name>SugarCRM Inc.</name>
    <homepage>
      <a href="http://www.sugarcrm.com/crm/about/about-sugarcrm.html">
        http://www.sugarcrm.com/crm/about/about-sugarcrm.html
      </a>
    </homepage>
    <icon path="images/icon.png"/>
  </vendor>
  <packager>
    <name>Parallels</name>
    <homepage>http://parallels.com</homepage>
    <uri>uuid:714f0a7b-85d6-4eb8-b68e-40f9acbb3103</uri>
  </packager>
  <presentation>
    <summary>Sugar CRM Community Edition</summary>
    <description>
      Sugar CRM Community Edition enables organizations to
      efficiently organize, populate, and maintain
      information on all aspects of their customer
      relationships. It provides integrated management of
      corporate information on customer accounts and contacts,
      sales leads and opportunities, plus activities such as
      calls, meetings, and assigned tasks. The system
      seamlessly blends all of the functionality required to
      manage information on many aspects of your business
      into an intuitive and user-friendly graphical interface.
    </description>
    <icon path="images/icon.png"/>
    <screenshot path="images/screen_home.png">
      <description>Home Screen</description></screenshot>

```

```

        <screenshot path="images/screen_admin.png">
<description>Admin Screen</description></screenshot>
        <screenshot path="images/screen_dashboard.png">
<description>Dashboard View</description></screenshot>
        <screenshot path="images/screen_accounts.png">
<description>Account Details</description></screenshot>
        <screenshot path="images/screen_campaigns.png">
<description>Marketing Campaigns</description></screenshot>
        <screenshot path="images/screen_bug_tracker.png">
<description>Bug Tracker</description></screenshot>
        <screenshot path="images/screen_calendar.png">
<description>Calendar</description></screenshot>
        <screenshot path="images/screen_documents.png">
<description>Documents</description></screenshot>
        <changelog>
            <version version="5.2.0a" release="1">
                <entry>Packaged as APS 1.1</entry>
            </version>
        </changelog>
        <categories>
            <category>
                Back office/Customer Relationship Management
            </category>
        </categories>
        <languages>
            <language>en</language>
        </languages>
</presentation>
<patch match="/application/version = '5.0.0'
or /application/version = '5.1.0a'"/>
<service id="instance">
    <license must-accept="true">
        <free/>
        <text>
            <name>GPLv3</name>
            <file>htdocs/LICENSE.txt</file>
        </text>
    </license>
    <presentation>
        <name>SugarCRM Instance</name>
        <summary>Basic services</summary>
        <entry-points>
            <entry class="control-panel" dst="/index.php"
method="POST">
                <label>Application entry point</label>
                <variable name="module">Users</variable>
                <variable name="action">Authenticate
</variable>
                <variable name="return_module">Users
</variable>
                <variable name="return_action">Login
</variable>
                <variable name="cant_login"/>
                <variable name="login_module"/>
                <variable name="login_action"/>
                <variable name="login_record"/>
                <variable name="user_name" class="login"
value-of-setting="admin_name"/>
                <variable name="user_password"
class="password"
value-of-setting="admin_password"/>
            </entry>
        </entry-points>
    </presentation>
</service>
</application>

```

```

        <variable name="login_theme">Sugar
    </variable>
        <variable name="login_language">en_us
    </variable>
        <variable name="Login">++Login++
    </variable>
    </entry>
</entry-points>
</presentation>
<!-- For each service setting declared in package, the corresponding
environment variable SETTINGS_[id] MUST be passed on to
the configuration script -->
<settings>
    <group class="authn">
        <name>Administrator's Account</name>
        <!--
SETTINGS_admin_name
-->
        <setting id="admin_name" type="string"
default-value="admin" min-length="1"
max-length="32"
regex="^[a-zA-Z][0-9a-zA-Z_\-]*"
class="login">
            <name>Administrator's Login</name>
            <error-message>Please make sure the text
you entered starts with a letter and
continues with either numbers, letters,
underscores or hyphens.
            </error-message>
        </setting>
        <!--
SETTINGS_admin_password
-->
        <setting id="admin_password" type="password"
class="password">
            <name>Administrator's Password</name>
        </setting>
    </group>
    <group class="web">
        <!--
SETTINGS_title
-->
        <setting id="title" type="string"
default-value="SugarCRM" class="title">
            <name>System Name</name>
            <description>This name will be displayed
in the browser title bar when users visit
the Sugar application.</description>
        </setting>
        <!--
SETTINGS_send_usage_statistics
-->
        <setting id="send_usage_statistics"
type="enum" default-value="true"
installation-only="true">
            <name>Send Anonymous Usage Statistics
            </name>
            <description>If selected 'Yes', Sugar
will send anonymous statistics about
your installation to SugarCRM Inc.
every time your system checks for

```

```

        new versions.
    </description>
    <choice id="true">
        <name>Yes</name>
    </choice>
    <choice id="false">
        <name>No</name>
    </choice>
</setting>
<!--
SETTINGS_check_for_updates
-->
    <setting id="check_for_updates" type="enum"
    default-value="automatic">
        <name>Check For Updates</name>
        <description>How the system will check
        for updated versions of the application.
    </description>
        <choice id="automatic">
            <name>Automatic</name>
        </choice>
        <choice id="manual">
            <name>Manual</name>
        </choice>
    </setting>
</group>
</settings>
<requirements xmlns:php="http://apstandard.com/ns/1/php"
    xmlns:db="http://apstandard.com/ns/1/db"
    xmlns:apache="http://apstandard.com/ns/1/apache">
    <!--
    PHP_VERSION
    -->
    <php:version min="5.1.0"/>
    <php:extension>mysql</php:extension>
    <php:extension>mbstring</php:extension>
    <!--
    DB_main_TYPE
    DB_main_NAME
    DB_main_LOGIN
    DB_main_PASSWORD
    DB_main_HOST
    DB_main_PORT
    DB_main_VERSION
    DB_main_PREFIX
    -->
    <db:db>
        <db:id>main</db:id>
        <db:default-name>sugarce</db:default-name>
        <db:can-use-tables-prefix>true
    </db:can-use-tables-prefix>
    <db:server-type>mysql</db:server-type>
    <db:server-min-version>4.1.2
    </db:server-min-version>
    </db:db>
</requirements>
<provision>
<!-- For each mapping, except ones which do not map to the
file system, WEB_<id>_DIR must be passed with the absolute
path to the directory to which mapping maps, where id is
the full URL path of the mapping, with all '/' characters

```

```

converted to '_'.
-->
        <url-mapping>
        <default-prefix>sugarcrm</default-prefix>
        <installed-size>53547008</installed-size>
        <!--
        WEB__DIR
        -->
        <mapping url="/" path="htdocs"
        xmlns:php="http://apstandard.com/ns/1/php">
        <php:handler>
        <php:extension>php</php:extension>
        </php:handler>
        <!--
        WEB__cache_DIR
        -->
        <mapping url="cache">
        <php:permissions writable="true"/>
        </mapping>
        <!--
        WEB__custom_DIR
        -->
        <mapping url="custom">
        <php:permissions writable="true"/>
        </mapping>
        <!--
        WEB__data_DIR
        -->
        <mapping url="data">
        <php:permissions writable="true"/>
        </mapping>
        <!--
        WEB__modules_DIR
        -->
        <mapping url="modules">
        <php:permissions writable="true"/>
        </mapping>
        <!--
        WEB__tmp_DIR
        -->
        <mapping url="tmp">
        <php:permissions writable="true"/>
        </mapping>
        <!--
        WEB__config.php_DIR
        -->
        <mapping url="config.php"
        virtual="virtual">
        <php:permissions writable="true"/>
        </mapping>
        </mapping>
    </url-mapping>
    <configuration-script name="configure">
        <script-language>php
        </script-language>
    </configuration-script>
</provision>
<service id="account">
    <presentation>
        <name>SugarCRM Account</name>
        <entry-points>

```

```

        <entry class="control-panel"
            dst="/index.php"
            method="POST">
            <label>Account entry point</label>
            <variable name="module">
                Users
            </variable>
            <variable name="action">
                Authenticate
            </variable>
            <variable name="return_module">
                Users
            </variable>
            <variable name="return_action">
                Login
            </variable>
            <variable name="cant_login"/>
            <variable name="login_module"/>
            <variable name="login_action"/>
            <variable name="login_record"/>
            <variable name="user_name"
                class="login"
                value-of-setting="user_login"/>
            <variable name="user_password"
                class="password"
                value-of-setting="user_password"/>
            <variable name="login_theme">Sugar
            </variable>
            <variable name="login_language">
                en_us
            </variable>
            <variable name="Login">++Login++
            </variable>
        </entry>
    </entry-points>
</presentation>
<!-- For each service setting declared in package,
the corresponding environment variable SETTINGS_[id]
MUST be passed on to the configuration script -->
<settings>
    <group class="authn">
        <name>Account Preferences</name>
        <!--
SETTINGS_user_login
OLDSETTINGS_user_login
-->
        <setting id="user_login" class="login"
            track-old-value="true"
            type="string" min-length="3"
            max-length="60"
            regex="^[a-zA-Z][0-9a-zA-Z_-]*">
            <name>Account's Login</name>
        </setting>
        <!--
SETTINGS_user_password
-->
        <setting id="user_password"
            class="password" type="password"
            min-length="4">
            <name>Account's Password</name>
        </setting>
    </group>
</settings>

```

```

</group>
<group class="vcard">
  <group class="fn n">
    <!--
    SETTINGS_first_name
    -->
    <setting id="first_name"
      class="given-name" type="string"
      max-length="30">
      <name>First Name</name>
    </setting>
    <!--
    SETTINGS_last_name
    -->
    <setting id="last_name"
      class="family-name" type="string"
      max-length="30">
      <name>Last Name</name>
    </setting>
  </group>
  <group class="email">
    <!--
    SETTINGS_user_email
    -->
    <setting id="user_email"
      class="value"
      type="email">
      <name>Email</name>
    </setting>
  </group>
  <!--
  SETTINGS_title
  -->
  <setting id="title" class="title"
    type="string">
    <name>Title</name>
  </setting>
  <!--
  SETTINGS_department
  -->
  <setting id="department"
    class="organization-unit"
    type="string">
    <name>Department</name>
  </setting>
  <group class="tel">
    <name class="type">work</name>
    <!--
    SETTINGS_phone_work
    -->
    <setting id="phone_work"
      class="value"
      type="string">
      <name>Work Phone Number
      </name>
    </setting>
  </group>
  <group class="tel">
    <name class="type">cell</name>
    <!--
    SETTINGS_phone_mobile

```

```
-->
    <setting id="phone_mobile"
    class="value"
    type="string">
        <name>Mobile Phone Number
        </name>
    </setting>
</group>
<group class="tel">
    <name class="type">fax</name>
    <!--
SETTINGS_phone_fax
-->
    <setting id="phone_fax"
    class="value"
    type="string">
        <name>Fax Number</name>
    </setting>
</group>
<group class="tel">
    <name class="type">home</name>
    <!--
SETTINGS_phone_home
-->
    <setting id="phone_home"
    class="value"
    type="string">
        <name>Home Phone Number
        </name>
    </setting>
</group>
<!--
SETTINGS_address_street
-->
    <setting id="address_street"
    class="street-address" type="string">
        <name>Street</name>
    </setting>
<!--
SETTINGS_address_city
-->
    <setting id="address_city"
    class="locality"
    type="string">
        <name>City</name>
    </setting>
<!--
SETTINGS_address_state
-->
    <setting id="address_state"
    class="region"
    type="string">
        <name>Region</name>
    </setting>
<!--
SETTINGS_address_country
-->
    <setting id="address_country"
    class="country-name" type="string">
        <name>Country</name>
    </setting>
```

```
<!--  
SETTINGS_postalcode  
-->  
  <setting id="address_postalcode"  
    class="postal-code" type="string">  
    <name>Postal Code</name>  
  </setting>  
<!--  
SETTINGS_description  
-->  
  <setting id="description" class="note"  
    type="string">  
    <name>Description</name>  
  </setting>  
</group>  
</settings>  
<provision>  
  <configuration-script name="usermanager">  
    <script-language>php  
  </script-language>  
    <status-control/>  
  </configuration-script>  
</provision>  
</service>  
</service>  
</application>
```

Index

A

About Application Packaging Standard • 4
About This Guide • 5
Adding and Removing Application Packages • 14
Appendix. Sample APP-META.xml • 59
Application Data Retention • 51

B

Backing Up Service Instance • 52

C

Concepts • 8
Configuration Script • 33
Configuring Application Global Settings • 19
Configuring Application Instance • 48
Controller Perspective • 9

D

Deploying Application Files • 26
Determining and Satisfying Requirements • 27

E

Enabling and Disabling Service • 50

F

Feedback • 7
Functions • 12

G

Getting Resources Usage Report • 56

I

Implementing Operations on Application Instances • 22
Implementing Operations on Application Packages • 13
Installing Service License • 29
Introduction • 4

M

Migrating Service Instance • 53

P

Patching and Upgrading an Instance • 42

Possible Scenario • 17, 20, 35, 39, 44, 49, 51, 54, 57

Processing Service License • 26
Processing Settings • 30
Provisioning Application Instances • 25
Provisioning Service • 32

R

Removing Application Instances • 38
Resolving Update Packages • 41
Restoring Service Instance • 53

T

Typographical Conventions • 6

U

Updating Application Instances • 40
URL Mapping • 34
Useful Links • 6