Introduction

The Provisioning Server is a system that can be used to provision software to OSGi based (and other) targets. It manages the life cycle of bundles and provides a controlled, centralized way to install, update and uninstall software and related artifacts (such as configuration data, native code and device firmware).

Purpose and scope

The purpose of this document is to provide a comprehensive overview of the software architecture of the system. It uses a number of different views to depict the important aspects of the system and captures and conveys the significant architectural decisions which have been made on the system.

It serves three purposes:

1. **Abstraction of the system.** The architecture provides a relatively small, intellectually graspable model of how the system is structured and how components interact.
2. **Mutual communication.** The model described here forms the common basis for all of the stakeholders to communicate with each other and form consensus about the system.
3. **Major design decisions.** The description of the architecture represents the earliest set of design decisions that have a significant impact on the system as a whole. They are relevant to ensure the qualities and features of the architecture.

Readers' guide

The document starts with an introduction, outlining the scope and purpose of this document and defining some acronyms and abbreviations. It then goes on by first sketching the architectural context, exploring the domain and the constraints. That is followed by the architectural design, which uses the 4+1 view as a guide to describe the system.

All stakeholders should read the architectural context, as it explains what the system does and how it interacts with its surroundings.

Software engineers in particular should read the architectural design, which outlines the foundation and describes the high level design that forms the basis for further analysis and design.

Definitions, Acronyms and Abbreviations

<table>
<thead>
<tr>
<th>Definition</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSGi</td>
<td>The OSGi alliance is an independent non-profit organization that maintains the OSGi standard. OSGi technology provides a service-oriented, component-based environment for developers and offers standardized ways to manage the software life cycle.</td>
</tr>
<tr>
<td>Provisioning</td>
<td>Software provisioning is the process of installing and updating software.</td>
</tr>
<tr>
<td>Gateway</td>
<td>A gateway, or OSGi gateway, is a computer or device that has an OSGi framework installed.</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer (REST) is a style of software architecture for distributed hypermedia systems. The term was introduced in the doctoral dissertation in 2000 by Roy Fielding, one of the principal authors of the Hypertext Transfer Protocol (HTTP) specification, and has come into widespread use in the networking community.</td>
</tr>
<tr>
<td>JMX</td>
<td>Java Management Extensions provides the tools for building distributed, web-based, modular and dynamic solutions for managing and monitoring devices, applications and service-driven networks. Since Java 5, it is part of the Java SE platform.</td>
</tr>
</tbody>
</table>
SOAP

SOAP is a protocol for exchanging XML-based messages over computer networks, normally using HTTP/HTTPS. SOAP forms the foundation layer of the Web services stack, providing a basic messaging framework that more abstract layers can build on.

DIY

An acronym for “do it yourself”.

References

This subsection contains an overview of all references used throughout this document. It should link directly to these references wherever possible.

<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>..</td>
</tr>
<tr>
<td>#</td>
<td>..</td>
</tr>
</tbody>
</table>

Architectural Context

Domains

The provisioning server consists of the following domains, as shown in the picture below. On the left hand side, the user interface and shop are shown. The shop is mainly concerned with the dependency management domain, effectively linking components to targets through various mechanisms of grouping and filtering based on licenses and capabilities of the individual bundles. On the right hand side, the provisioning part deals with the actual deployment of versions of components to targets. Finally, on the target itself, the life cycle is monitored and managed. The bundle repository does not need to be a part of our system, but it is used both in the shop and during deployment.

Deployment

In short, deployment deals with getting software components onto target systems. The general strategy is to have a management agent on each OSGi based target that receives and deploys these components.

An important aspect of deployment is the actual distribution of components. The provisioning server takes a bundle repository as its source and bundles somehow need to find a way to the targets. In real-life scenarios, there often won’t be a completely open, two way connection between server and target, so catering for all kinds of scenarios here is important.

Life Cycle Management

Life cycle management deals with managing the life cycle of the individual bundles within the OSGi based target. For these bundles, the following states can be identified:

- installed;
- resolved;
- starting;
The following state diagram shows these states and their transitions:

![State Diagram]

The transitions are explained below:

- **install** - Each bundle starts its life cycle when it is first installed in the OSGi framework. When a bundle is installed it is stored persistently in the framework.
- **start** - As soon as the bundle is started, it will transition through a couple of states. The first step is the resolving of package dependencies. Here the bundle is "wired up" and if that succeeds, it ends up in the resolved state. From there it will go to starting, where the bundle activator gets instantiated and a bundle can become an active entity (it can start threads, initialize, etc.). Finally it ends up in the active state.
- **update** - As soon as a bundle is installed, it can be updated. When a bundle is updated, if it was active, it will be stopped. Subsequently it will have to be resolved again, and started.
- **stop** - When a bundle is active, it can be stopped. It will first go to the stopping state, where it will have to cleanup (basically undo everything it did during starting). It will end up as resolved.
- **uninstall** - When a bundle is no longer needed, it can be uninstalled. That's a final state, from there it can never be started again.

### Dependency Management

Whenever you are dealing with collections of bundles that are installed together, you need to make sure this collection actually works together. Each bundle can have dependencies on other bundles, services, packages or even specific hardware or operating systems. These dependencies all need to be managed.

Within an OSGi framework, there are two layers that feature dependencies:

1. the module layer, that has package dependencies;
2. the service layer, that has service dependencies.

Traditionally, in OSGi a bundle contains enough meta-data to analyze package dependencies and ensure that these can be resolved. Service dependencies, however, are a lot harder to analyze because of the extremely dynamic nature of a service and the fact that there is no meta-data available.

For other dependencies, such as required screen sizes, or the presence of specific hardware, no meta-data is available in the bundle, so that is one thing we need to add externally.

Summing it up, it is important to make sure that you end up deploying sets of bundles that work together well in the environments in which they’re deployed.

### Limitations

About the limitations of the chosen architecture...

1. ...
2. ...

### Context

...

### Constraints
Whitebox analysis

Architectural Design

The architecture is described using the 4+1 view, where architecturally relevant use-cases describe the main features of the system and logical, process, deployment and implementation views explain how these features are implemented.

Architectural foundation

Service oriented, component based architecture

The architectural style we choose for implementing the provisioning server is a service oriented, component based architecture, based on the OSGi framework. This section outlines the reasons for using this style and explains its important strong-points:

1. Clean separation between interface and implementation. Allows us to even provide different implementations of the same service.
2. Loose coupling between components. Allows us to partition components based on the actual deployment scenario. We have many different deployment scenarios, so by having components we can assemble them in different ways depending on the scenario.
3. Easy integration. Both on the component level and when providing different remoting protocols.

Principles

Object orientation and Java

We use Java and OO throughout the implementation. Java gives us platform independence and is available on our "server", "client" and "target" deployment nodes. We use different Java versions for each of them:

- The target will stick to the standard OSGi execution environment available, so it's compatible to all OSGi environments. This is the CDC-1.1/Foundation-1.1 environment as specified in OSGi R4.1 compendium chapter 999, which in turn refers to http://java.sun.com/products/foundation for a full definition.
- The server will use Java 5, because it's widely available, contains a lot of interesting new features over older VM's and has a consistent memory model.
- The client will use Java 6 on the desktop, because there was a considerable focus on this platform from Sun which makes this version a lot better than Java 5 here. Java 6 runs on all platforms we intend to support for the desktop (Windows, Linux and Mac OS X). In the past we did use Java 5 and some of the problems we ran into are described in the Swing UI document.

Compose instead of build

We do not want to create different builds, but have components and user settings to tailor the system. By having a loosely coupled system that uses services, we can create relatively simple components that we can compose in different ways. At the same time, we can build on existing services.

Partition data so it is changed in one place

Merging data that was created in more than one place is difficult, so we try to avoid that. Having multiple repositories to partition the data in meaningful ways is the solution. These repositories can then be replicated elsewhere, but those copies will always be read-only, sometimes plain copies of the original, other times aggregates of a set of non-overlapping repositories.

Use standards based software whenever possible

2. OSGi. Suits the service oriented style well. Proven technology.
3. JMX. Used to integrate with management systems.
4. Certificates. Used to identify all actors (systems and users).

Eat our own dogfood

As a software provisioning system, we need to make sure that we can provision ourselves too. This will be an important showcase for our own product. Both bootstrapping and updating should be painless.

Use Case View

The roles document describes the different roles within the system. Together with the Use Cases document, the functional requirements of the system are described. The architecture document describes the subset of architecturally relevant use cases. Use cases can be architecturally relevant because they exercise many architectural elements, or if they illustrate a specific, delicate point of the architecture. These use cases are:
• Resolve merge conflict, because it describes the most complex behavior when multiple people are editing the same repository, and we need to be clear about how these cases are handled and how the users experience it.
• Check for update, because this is the core mechanism for gateways to get new (versions of) software and in the end, that's what provisioning is all about.
• Approve version, because it is an important (logical) step in the whole process and is basically the point where you decide what new set of artifacts to publish to a gateway.

Use-Case Realizations

Here, the realizations of the use cases mentioned in the previous paragraph are described. Use case realizations in general describe how a particular use case is realized within the design model: it describes how objects collaborate.

Resolve merge conflict
This is a realization of UC Resolve merge conflict.

Check for update
This is a realization of UC Check for update.

Approve version
This is a realization of UC Approve version.

Logical View

Overview

Architecturally Significant Design Packages

Management agent

The management agent consists of the following components:

1. Discovery, which gives the agent the ability to find the provisioning or relay server. We can have more than one implementation of this, depending on the customers' needs:
   a. manual discovery, which means a "static" server is configured (which is what we have now...)
   b. "rendezvous" discovery
   c. DIY multicast discovery
   d. brute force network scan based discovery
2. Control, a communication channel that can be used to pass control information. We are still looking for really lightweight protocols for this, or we might just use HTTP.
3. Deployment, the actual data channel or stream, which is covered by the Deployment Admin specification.
4. Identification, the strategy that is used to identify the gateway. We currently already support a number of strategies.
5. Cache, which can store “old” versions of deployment packages and bundle data, if there is enough space on the device. This is of course an optional component.
6. Audit log, logging all actions on the gateway. Separate component because we might want to have multiple implementations, depending on device capabilities.

Aspects:
1. Logging (solved by using the OSGi log service)
2. Managability (JMX based...)
3. Configuration (Config Admin based...)
4. Security (probably we won’t even run without it, based on certificates, ...)

Repository
Within the system, there are several repositories. Since the repository is such a key concept, it is described here. We start by identifying the different types of repositories. Then, for each type, we look in more detail at the information it contains.

By examining the domains in the architectural context we can see we need several repositories. When we look at provisioning, more specifically the deployment and life cycle management domains, we see two repositories there: the bundle repository and the audit log. If we include the dependency management domain, we see the need for additional repositories to manage the relations between bundles and gateways, and the grouping of bundles. That requires several object repositories. For a detailed analysis, read the repository analysis which explains in more detail how these types of repositories will be designed and implemented.

In the next paragraphs, we will go into more detail for each type of repository.

Bundle repository
As the bundle repository analysis explains, it makes sense to map this repository to the existing OBR standard here (RFC-112). This allows us to use existing repositories, as well as our own ones.

Audit log
The audit log is a repository that is maintained on each individual target, as explained in the audit log analysis. Audit logs are subsequently replicated and aggregated to make them available elsewhere without needing a direct connection to the gateway you’re examining.

The management agent, on the target, also marks if the target is in “special” mode or not. There are two variations of this special mode:

1. Fixed. Here the gateway is fixed to an old version because of a problem that was noticed by the gateway operator while or after an update.
2. Unmanaged. Here the gateway has custom software that was deployed by the gateway operator and is not yet available in the shop. It is the responsibility of the operator to consolidate those changes when he returns to the server.

These modes are also recorded to the audit log.

Object repository
You can look at object repositories at two levels. At the lower level, you see an object repository as a store where you can keep versioned copies of “blobs”. You don’t actually need to know the semantics of these “blobs”. At that level, the object repository analysis describes how to deal with these repositories. In short, each object repository:

- is versioned, where each change updates the version number;
- can be accessed stream based with a simple checkout/commit style interface;
- has only one physical location where you are allowed to change data (the master copy).

Object repositories can be replicated (many times). Each object repository should have one “master” copy, and that’s the only location where changes can be made. All (partial) copies are read-only.
At the higher level, you are actually interested in what's inside these "blobs". A separate repository model analysis describes this.

The following repositories exist:

1. The "store", which contains components, groups and licenses.
2. The "license manager", which contains licenses and targets. In a lot of cases, this component will be connected to some kind of external license manager that determines exactly how and why certain targets get certain licenses.
3. The "gateway operator" repository, which contains targets, versions and bundles. Versions can have a state (new, approved, ...).
Remoting
There are many ways in which the OSGi service repository can be extended to support remote method calls. The generic way of doing this is by adding specific (marked or annotated) interfaces that implement these remoting aspects.

REST
To communicate between nodes in our topology, we use a REST based API.

SOAP
To allow external systems to communicate with us, we use SOAP.

JMX
To make our system manageable, we use JMX.

Security
Security is based on certificates. Certificates identify both humans and computers. A central "root" certificate server hands out all identities. Additional "child" servers can be used to hand out certificates based on that root certificate, or any other certificate coming from the same root.

Client
A client can be used to manipulate repositories. Clients do a "checkout" of repository data, change things and then commit back those changes. Advanced clients can also deal with more complex "merge" scenarios.

Process View

Server
Several active components exist:

1. Listener for incoming poll requests.
2. Listener for repository commands (checkout, commit).
4. ...many more...⚠️ TODO

Target
A couple of strategies exist for the target:

1. Passive, where an external scheduler is used to initiate a poll. This can be used when the gateway itself knows when it has free CPU cycles to check for and/or install updates.
2. Active, where a built-in scheduler initiates a poll, which will probably be done periodically or according to some specific schedule.
3. Reactive, when some external system notifies the gateway. This can either be a server that knows there's an update, or it can be someone who signals it's time to do a poll.

Deployment View
The following types of nodes make up the deployment view:

- Client
- Provisioning Server
- Bundle Repository
- Relay Server
- Target
For a product such as this, there is no single deployment view, since the deployment topology can be adapted to the requirements of a specific customer. What we can do here is show a couple of different deployment scenarios. The main point to be made here is that all components that make up the system should be loosely coupled to allow easy deployment in all these scenarios.

## Simplest possible system

The simplest possible system is one where there is only one node that runs all provisioning software. This node can either be a server, stand-alone workstation or even a laptop. It contains everything and communicates with the gateways directly.

- Server (runs the client, provisioning server and bundle repository)
- Target

## Hosted system

A hosted system is one where the provisioning server software is hosted on the internet. There still are a couple of scenarios here. This one describes a customer that does not want its intellectual property to be available on the internet anywhere. That means that both the bundle repository and the relay server(s) are on its "local network" (which might be any type of network outside the internet).

- Client(s)
- Hosted provisioning server
- Local relay server(s)
- Local bundle repository
- Target

## Regional system

This is a system where software is developed in one location, but each regional market has its own release and licensing policies. Here, each relay server manages gateways for one region. The bundle repository might be replicated for each region.

- Client(s)
- Provisioning server
- Relay servers
- Target

## Implementation View

### Overview

### Layers

### Size and Performance

### Quality