

PowerMatcher Core

Installation and Configuration

Authors:

A.H. Eisma (IBM)
C. Haverkamp (IBM)

Release: **0.9**

Revision: **2**

Date: 14-2-2013



TABLE OF CONTENTS

1.	Introduction	7
2.	PowerMatcher Core overview	8
2.1	Core components	8
2.2	Adapters	8
2.3	Run-time environments	9
2.4	Component configuration	10
2.5	Manual Configuration in OSGi using the Web Console	10
2.6	Configuration using XML	12
2.6.1	Concept	12
2.6.2	Processing the node configuration	14
2.6.3	Configuration XML	14
2.6.4	Configuration and Property elements	15
2.6.5	Node Configuration	16
3.	PowerMatcher stand-alone application configuration	17
3.1	Configuration using property files	17
3.2	Configuring adapter factories	17
3.3	Configuring components	17
3.4	Running the stand-alone application	18
4.	Creating a PowerMatcher configuration	19
4.1	OSGi configuration example	19
4.1.1	Manual configuration using the OSGi web console	19
4.1.2	Configuration in the OSGi runtime using XML	21
4.2	Configuration of a stand-alone Java runtime with a property file	22
5.	Development Environment Setup	24
5.1	PowerMatcher maintenance versus application development	24
5.2	Install a Java JDK	24
5.3	Eclipse download and install and setup	24
5.4	Installing the Bndtools plug-in	25
5.5	MQTT broker installation	26
5.6	Retrieving the source code	26
5.7	IDE Setup	26
5.8	Importing the PowerMatcher Core binary distribution	29
5.9	Importing the PowerMatcher Core and Core Example sources	29
5.10	Code style and formatting preferences	31
5.11	Launching the application from the workbench	31
5.11.1	Run as a stand-alone Java application	31
5.11.2	Run as an OSGi application	32
5.12	Building a PowerMatcher Core application	32
5.12.1	Build the stand-alone Core application	32
5.12.2	Build the PowerMatcher Core OSGi bundle and library jars	34
6.	Appendix A: PowerMatcher Components	36
6.1	Core components	36
6.1.1	Auctioneer	36
6.1.2	Concentrator	37
6.1.3	CSV Logging agent	38
6.1.4	Objective agent	38
6.1.5	Test agent	40
6.1.6	Template agents	41
6.2	Core Component Adapters	42
6.2.1	Market Basis Adapter Factory	42
6.2.2	Direct Protocol Adapter Factory	42
6.2.3	Agent Protocol Adapter Factory	42
6.2.4	Matcher Protocol Adapter Factory	43
6.2.5	Direct Logging Adapter Factory	43
6.2.6	Logging Adapter Factory	43
6.2.7	Log Listener Adapter Factory	44
6.2.8	MQTT v3 Connection Adapter Factory	44
6.2.9	Time Adapter Factory	45
6.2.10	Scheduler Adapter Factory	45
6.3	Configuration Manager component	45

7.	Appendix B: OSGi runtime installation and setup	47
7.1.1	Pax Runner installation and setup	47
7.1.2	Install the prerequisite bundles	48
7.1.3	Install the PowerMatcher bundles.....	48
7.1.4	Configure the run time properties	49
7.1.5	Logback logging configuration	49
7.1.6	Install and run the MQTT broker.....	50
7.1.7	Configure the PowerMatcher application.....	50
7.1.8	Starting the OSGi environment.....	50
8.	Appendix C: Configuration XML schema.....	51
9.	References	53

TABLE OF FIGURES

Figure 1 The Configuration tab in the Apache Felix Web Console.	11
Figure 2 The dialog for creating or editing configuration records.	12
Figure 3 Example configuration specification.	14
Figure 4 Creating an Auctioneer configuration record in the Apache Web Console.	20
Figure 5 The Eclipse dialog for selecting an existing or new workspace.	24
Figure 6 The Eclipse workbench after creating a new workspace.	25
Figure 7 Installing the Bndtools plugin from the Eclipse Marketplace.	25
Figure 8 JRE definition and selection in the Eclipse Preferences window.	26
Figure 9 Bndtools wizard for creating a configuration project named 'cnf'.	27
Figure 10 Select the Default Configuration template for the 'cnf' project.	28
Figure 11 The 'cnf' project created by Bndtools prefilled with common bundles.	28
Figure 12 Importing existing projects using the Import Projects wizard of Eclipse.	30
Figure 13 The user library definitions in the Eclipse preferences window.	30
Figure 14 Eclipse preferences import wizard.	31
Figure 15 Launching the PowerMatcher application from the Run-button in the Eclipse toolbar.	32
Figure 16 Selecting the 'Runnable JAR file' option from the Eclipse Export wizard.	33
Figure 17 Eclipse wizard for exporting a runnable JAR file.	33
Figure 18 Build Core bundles using the Bndtools Release Bundles wizard.	34
Figure 18 Build Core bundles with the build.xml script in the OSGi example project.	35

VERSION HISTORY

Version	Description	Author
0.7 - 1	Initial version for PowerMatcher release 0.7	C. Haverkamp (IBM)
0.9 - 1	Updated for PowerMatcher release 0.9	A.H. Eisma (IBM)
0.9 – 2	Updated for Bndtools 2.0	A.H. Eisma (IBM)

1. Introduction

The PowerMatcher Core offers a framework for developing PowerMatcher based systems. It provides all the base components of the PowerMatcher concept: auctioneer, concentrator, objective agent and interfaces and framework classes to develop your own PowerMatcher enabled device agents. For logging purposes the framework contains also a CSV Logging Agent and a Test Agent for demonstration purposes.

PowerMatcher systems can be developed as stand-alone applications or run on the OSGi platform. The latter is a runtime environment where the PowerMatcher components are deployed as modules and where each module provides services and may have dependencies on other modules or on configuration data.

Running your PowerMatcher system on the OSGi platform offers the possibility of dynamically configuring and updating your system and supports a distributed architecture. Configurations can be stored in XML files and can be updated while the system is operating. The PowerMatcher ConfigManager component running within OSGi provides regularly checks the configuration and updates the configuration objects in the OSGi runtime. OSGi will restart components with updated configuration settings automatically.

Systems initially designed as stand-alone PowerMatcher applications can be upgraded to the OSGi platform. It just requires implementing the OSGi component interface for you custom made device agent, the installation and set up of an OSGi runtime and creating a ConfigManager configuration version for your PowerMatcher application.

This document describes how to install and configure a PowerMatcher system and development environment. It provides an overview of the available components and adapters and their configuration properties. The different alternatives for configuring the system is explained by creating a sample configuration which can be the start of your own PowerMatcher based application.

2. PowerMatcher Core overview

2.1 Core components

The basis of a PowerMatcher application is formed by a set of components that will provide the core PowerMatcher functionality. This set consists of the following components:

- **Auctioneer**
The auctioneer component creates an auctioneer instance, which will receive all the bids of the other agents as a single bid or as an aggregate bid via one or more concentrators. It is responsible for defining and sending the market basis and calculating the equilibrium based on the bids from the different agents in the topology. This equilibrium is communicated to the agents down the hierarchy in the form of price update messages.
- **Concentrator**
The concentrator is a factory component from which several instances can be created. The network of agents can be divided into several groups of agents, each group having their own concentrator. The concentrator receives the bids from the agents and forwards this in an aggregate bid up in the hierarchy to a concentrator or to the auctioneer directly. It will receive price update messages from the auctioneer and forward them to its connected agents.
A special type of concentrator component, the Clipping Concentrator, can limit the power consumption of its group to a certain configurable level.
- **Objective agent**
This component influences the market price and the power consumption to a configurable value.
- **CSV Logging agent**
This agent subscribes to price update and bid log messages, sent by agents in the topology, and stores them in a comma separated file (csv-file).
- **Test agent**
Though the purpose of PowerMatcher Core is to provide a framework to design your own PowerMatcher based applications with custom developed device agents, it also contains a Test Agent for testing your initial setup.

All communication between the components is implemented via a service bus using the MQTT-protocol. In order to run a PowerMatcher environment an MQTT broker is required.

2.2 Adapters

The core components are accompanied by a set of adapter components that are required by the core components. The following adapters are provided:

- **Market basis adapter**
This adapter defines the market basis that on which all bid and price update messages are based upon. The adapter is used by the auctioneer only.
- **Agent Protocol adapter**
The capability to publish bids to and receive price updates from matchers (auctioneers and concentrators) via a messaging connection is provided by this adapter component. The communication protocol is configurable, but should match the protocol of the matcher.
- **Matcher Protocol adapter**
A matcher component (auctioneer, concentrator, objective agent) can receive bids from and publish price updates to its child agents through its Matcher Protocol adapter via a messaging connection. The communication protocol is configurable and should match the communication protocol of the Agent Protocol adapter of its connected agents.
- **Direct Protocol adapter**
This adapter lets an agent connect directly to its matcher in the same runtime environment via procedure calls. This is the most efficient and best scalable type of adapter.
- **MQTT connection adapter**
A messaging connection to the MicroBroker MQTT bus is provided by the MQTT connection adapter. There are two different adapters, a version 3 and a version 5 adapter, depending on the version of the MQTT protocol that is being used on the service bus.

- **Logging adapter**
Enables the agent or matcher component to log PowerMatcher bid and price updates sent or received via a messaging connection.
- **Log listener adapter**
Enables a logging component, like the CSV Logging Agent, to receive bid and price update log messages from a messaging connection.
- **Direct Protocol adapter**
This adapter lets an agent or matcher connect directly to its logging component in the same runtime environment via procedure calls. This is the most efficient and best scalable type of adapter.
- **Time and Scheduler adapter**
These adapters provide real or simulated time and thread-pool based scheduling services.

2.3 *Run-time environments*

There are two alternatives to run a PowerMatcher Core based application:

1. **Stand-alone Java application**
Running the PowerMatcher application as a Java application just requires a platform that has a java runtime (JRE, version 5 or higher) installed. The application components (auctioneer, concentrator, device agents etc.) will be configured using a property file.
2. **PowerMatcher on an OSGi runtime environment**
The PowerMatcher components are also supplied as bundles which can be deployed on an OSGi runtime environment. This requires an OSGi runtime environment, like for example Pax-Runner.

In both alternatives an MQTT broker is required if one or more messaging connections are configured. Examples of MQTT brokers are Micro Broker, which is shipped as part of the IBM Lotus Expeditor Client, or Mosquitto[11] an open source message broker. See <http://mqtt.org> for more information on MQTT.

No MQTT broker is required if all configured connections are direct connections.

2.4 Component configuration

The PowerMatcher runtime, both for the OSGi and the stand-alone Java runtime environment, is based on a component model where components have dependencies that need to be resolved before they can be activated. The runtime provides a dependency injection framework where components are wired together to resolve these dependencies.

A component, for example an auctioneer, cannot be created and activated before a number of dependencies are fulfilled. First the auctioneer itself needs a configuration object that provides all required properties. Next, the PowerMatcher runtime will create an auctioneer object instance, although this object still has a number of dependencies to be resolved before it can be activated.

One example of a dependency is the MQTT connection that is needed for an agent or matcher messaging protocol adapter. In order to communicate with the MQTT service bus the protocol adapter needs an MQTT connection adapter. In the PowerMatcher Core adapter framework, all adapters are created automatically from adapter factories using dependency injection.

- Each component identifies the adapter factory (by the combination of cluster ID and factory ID) that it uses for a particular type of connector that it provides.
- When a component is activated, the adapter framework locates the component's adapter factories and instantiates and configures the adapters and connects them to the component.
- When a component is deactivated, the framework disconnects and deletes the adapters.

The adapter framework creates adapters recursively. That is, if a component requires a protocol adapter, the framework creates the protocol adapter from the factory specified by the component. If the protocol adapter requires a connection, the framework creates the connection adapter from the factory specified by the protocol adapter.

For the OSGi runtime environment, creating a PowerMatcher Core application basically comprises the deployment of the PowerMatcher Core components as OSGi bundle jars in the OSGi runtime, deploying the custom made device agents and then create the configuration objects to instantiate and activate the components.

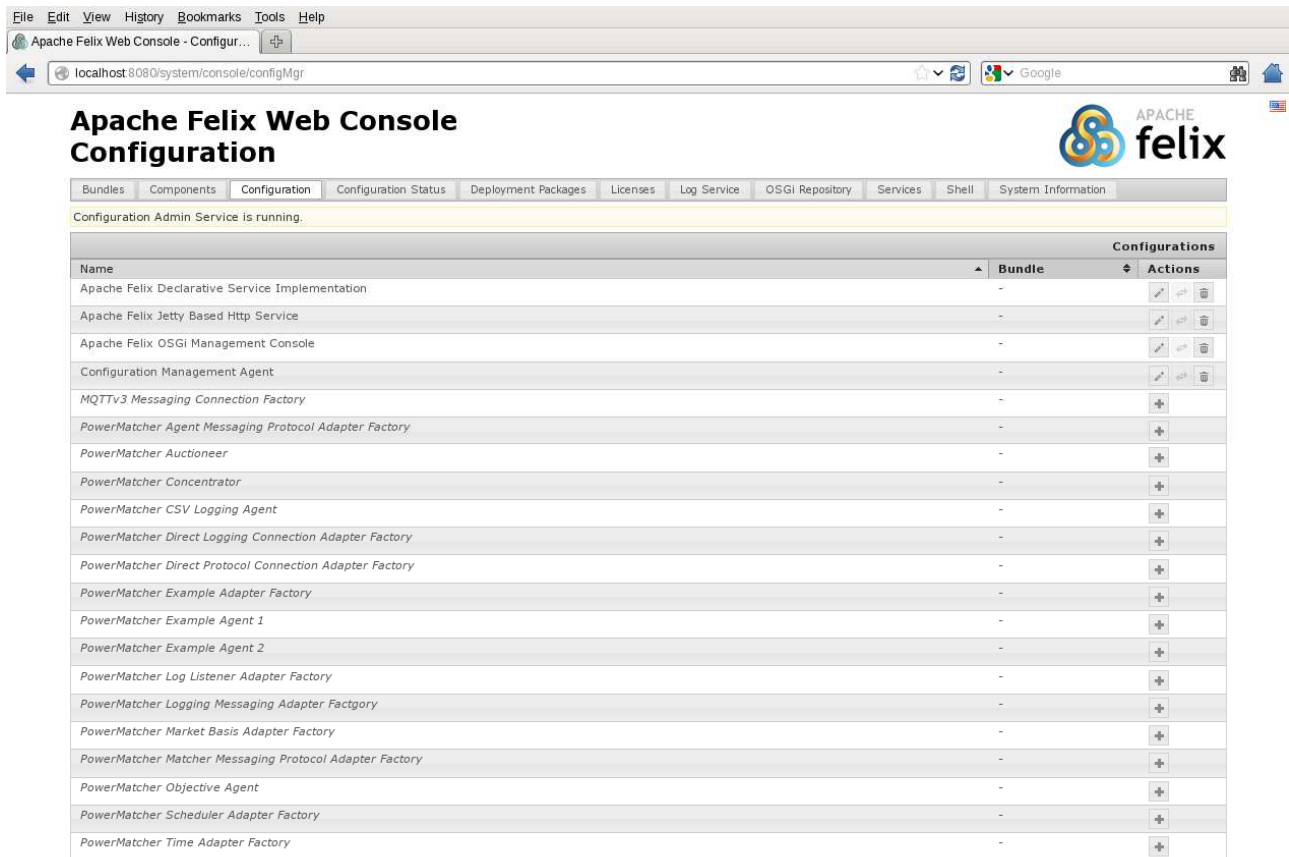
The following sections describe the alternatives to create these configuration objects.

2.5 Manual Configuration in OSGi using the Web Console

Using the Apache Felix Web Console for OSGi, a PowerMatcher configuration can be created manually. Each OSGi configuration for a component that is created with the web console is already prefilled with the default values.

The first step is to launch an unconfigured OSGi runtime containing the PowerMatcher Core (and optionally Extension) bundles, for example using the *PowerMatcher Core OSGi Empty Launch.bndrun* launcher in project *net.powermatcher.core.launcher.osgi.template*. This launch configuration starts the runtime with an empty configuration because the project does not contain the default configuration file *conf-local/default_config.xml*.

To create configurations for deployed components, open the Apache Felix Web Console (<http://localhost:8080/system/console>) using the default username *admin* with password *admin*, and navigate to the Configuration tab. Find the component's symbolic name in the list. Figure 1 shows the configuration tab. Note that it displays the configuration records that are already created below the symbolic name of the component. OSGi factory components have a '+' button for creating multiple configuration records and a singleton component will only have one configuration record, hence the edit button (the pencil).



The screenshot shows the Apache Felix Web Console interface. The browser address bar indicates the URL is `localhost:8080/system/console/configMgr`. The page title is "Apache Felix Web Console Configuration". The Apache Felix logo is in the top right corner. A navigation bar contains tabs: Bundles, Components, Configuration (selected), Configuration Status, Deployment Packages, Licenses, Log Service, OSGi Repository, Services, Shell, and System Information. A status message states "Configuration Admin Service is running." Below this is a table of configurations.

Name	Bundle	Actions
Apache Felix Declarative Service Implementation	-	[edit] [delete]
Apache Felix Jetty Based Http Service	-	[edit] [delete]
Apache Felix OSGi Management Console	-	[edit] [delete]
Configuration Management Agent	-	[edit] [delete]
MQTTv3 Messaging Connection Factory	-	[+]
PowerMatcher Agent Messaging Protocol Adapter Factory	-	[+]
PowerMatcher Auctioneer	-	[+]
PowerMatcher Concentrator	-	[+]
PowerMatcher CSV Logging Agent	-	[+]
PowerMatcher Direct Logging Connection Adapter Factory	-	[+]
PowerMatcher Direct Protocol Connection Adapter Factory	-	[+]
PowerMatcher Example Adapter Factory	-	[+]
PowerMatcher Example Agent 1	-	[+]
PowerMatcher Example Agent 2	-	[+]
PowerMatcher Log Listener Adapter Factory	-	[+]
PowerMatcher Logging Messaging Adapter Factory	-	[+]
PowerMatcher Market Basis Adapter Factory	-	[+]
PowerMatcher Matcher Messaging Protocol Adapter Factory	-	[+]
PowerMatcher Objective Agent	-	[+]
PowerMatcher Scheduler Adapter Factory	-	[+]
PowerMatcher Time Adapter Factory	-	[+]

Figure 1 The Configuration tab in the Apache Felix Web Console.

Click the '+' to create a new configuration record or the edit button for a single configuration record (e.g. for the Configuration Management Agent). The dialog that appears is already prefilled with default values (see the example in Figure 2). After you entered the properties click the Save button and the configuration record will be created. The Configuration Admin service will now instantiate component using the configuration data.

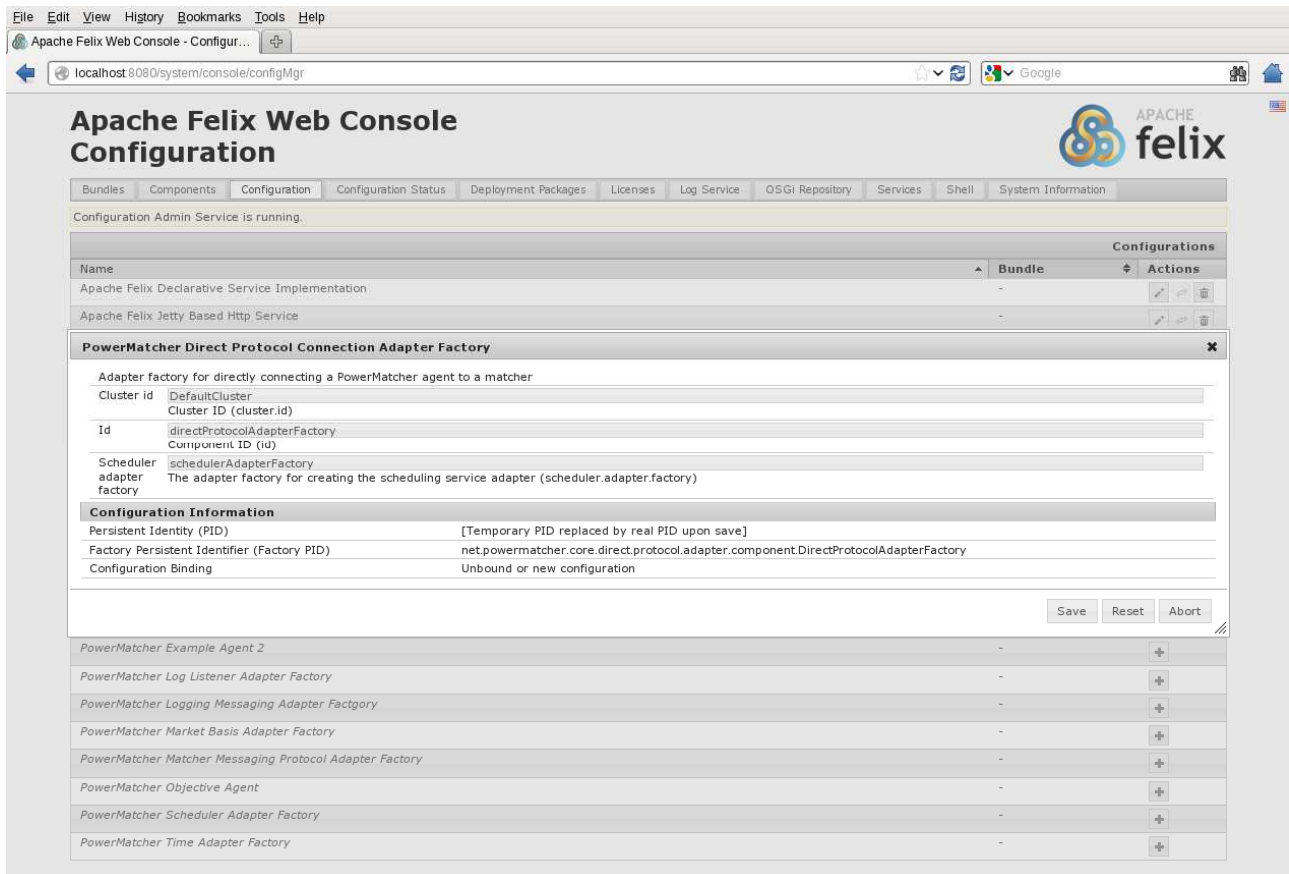


Figure 2 The dialog for creating or editing configuration records.

2.6 Configuration using XML

Specifying configurations using the Felix Web Console may be simple, when this procedure has to be repeated several times it is more sensible to do a configuration from a persistent resource like a file (note however that configurations in the OSGi runtime environment are persistent, but deploying a similar configuration on a different node requires creating a new configuration).

The PowerMatcher Core framework facilitates configuration of a complete PowerMatcher runtime environment using a single XML configuration file. This service is provided by the Configuration Manager Agent which is also an OSGi component itself. It processes the configuration specifications in the XML file and creates, updates or deletes configurations for the components by invoking the Configuration Admin service of the OSGi runtime.

2.6.1 Concept

The idea of the configuration of OSGi components is based around two main entities: *Configuration Specifications* and *Node Configurations*. The Configuration Specification defines the contents of an OSGi configuration object record. A node represents an OSGi runtime container instance that is the host of a set of components.

The Node Configuration defines the configuration records for this constellation of components in a node by linking to one or more Configuration Specifications. Each Configuration Specification can have multiple child configurations together forming a tree structure.

The actual values of the fields in the OSGi configuration record are defined using a third entity: a *Property* which contains a key-value pair where the key is the name of the field in the configuration record. A Configuration Specification can have multiple properties.

The **Configuration Specification** can have different roles and characteristics:

- **Singleton Component Specification:**
This type defines the configuration record of a singleton, component; an OSGi component from

which only one instance can be created. The component record is created with a *Persistent Identity* ("PID") that matches the components name. This will be also the identifier of the component instance in the OSGi container.

- **Factory Component Specification:**

OSGi components types from which multiple component instances can be created are called 'factory components'. The creation of configuration records will require in this case a *factory PID*, which is equal to the component name and an identifier which should be unique for only the configuration records of this component type (i.e. PID). Component Specifications of type factory are used to create this type of configuration records.

Note that all PowerMatcher adapters are created from adapter factories, and that the adapter factory itself is implemented as an OSGi factory component. So, whenever a new adapter factory component is created in OSGi, for example by pressing the + in the web console, a new adapter factory is activated that itself will automatically create adapters.

- **A configuration group:**

In this role the a Configuration Specification enables the logical grouping of Configuration Specifications. The configuration group can also define properties, which will be propagated to its child Configuration Specifications unless the child overrides that property.

- **Component Specification Template:**

In case many instances of the same component type need to be created, which share many properties having the same value, you can define a Configuration Specification that acts as a template. The properties of a template will be applied to a configuration when there is a template configuration with the same PID which is a child of one of the ancestors (parents) in the hierarchy of the template user.

Each node is running an OSGi container where a collection of components (or *bundles*) are deployed. The configuration for this node and its component instances is defined in the **Node Configuration**. However, the configuration of the components has to be done within the container, which means that each node requires also its own Configuration Management Agent.

Figure 3 demonstrates the configuration concept using a simple example. Suppose a company has a number of Windows and Linux servers that will be monitored by an agent that will be running in an OSGi container. There will be two agent types, a Windows Monitor Agent and a Linux Monitor Agent. From both types several agents will be instantiated, each monitoring one server. The Configuration Specification for these agent types will therefore be of type 'factory'. The configuration specifications of the same type are grouped into a Configuration Group. Because many Windows and Linux Monitor Agents share the same configuration parameters for each agent type there is a configuration template defined.

There is one singleton Configuration Specification, which is the configuration for the Security Agent that will monitor the Directory Services and other security aspects. Since there is only one agent required, the type of this Configuration Specification is 'singleton'.

The configuration the company's monitoring agents is defined by the 'Multi-system node', this configuration contains the Windows and Linux Monitor Agent configurations and the Security Agent configuration.

Suppose the company has two daughter companies one having only Windows servers, the other solely running Linux servers. Both have also monitor agents for their servers. The company running Windows servers can configure the agents with the 'Windows cluster node' configuration and the other can use the 'Linux Cluster Node' configuration. Since they share the network facility with their parent company they do not require a security agent. If they didn't the definition of the Linux and Windows cluster nodes could have also added the Security Agent configuration.

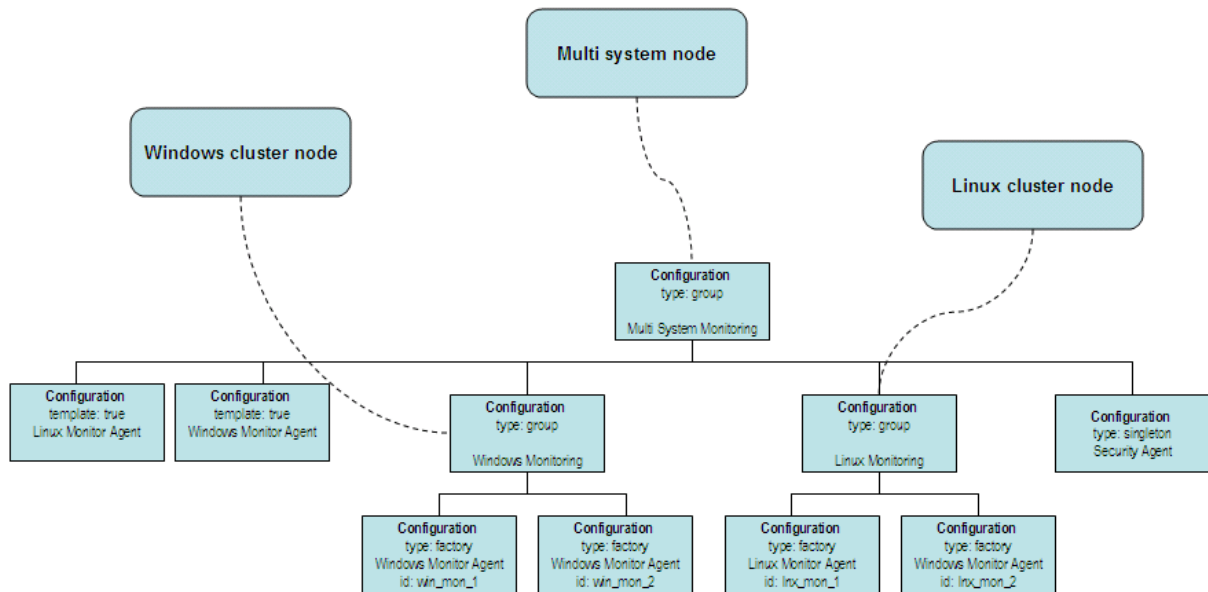


Figure 3 Example configuration specification.

2.6.2 Processing the node configuration

The Configuration Management Agent will inspect the node configuration. If the configuration is valid it will process it and invoke the Configuration Admin service with information requests about the current configuration records. In case a configuration specification has no matching configuration record in the container it will request the Configuration Admin service to create that configuration. If it already exists the properties of the runtime configuration instance are compared with the version in the node configuration. In case there are changes found, the configuration record will be updated. Configuration records that have no matching specification in the node configuration will be deleted.

The Configuration Management Agent implementation for PowerMatcher demands that each configuration specification has an 'id' and a 'cluster.id' property. The id is required to match runtime configuration records with their specification counterparts in the node configuration.

2.6.3 Configuration XML

The input of the Configuration Management Agent is a *node configuration*. This node configuration is an XML structure that is defined by a schema or XSD (see to section 8

Appendix C: Configuration XML schema).

The XML structure of contains elements that match the entities mentioned earlier in the description of the concept. A node configuration xml structure will contain the following elements:

- configuration
- property
- nodeconfig

The XML representation of a Node Configuration is a tree structure of configuration elements below the node. The following XML excerpt shows an example of a Node Configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<nodeconfig id="example-node" name="Example node"
  description="PowerMatcher example node" date="2012-08-29 16:02:06">
  ...
  <!-- ExampleCluster Core Example Agents Group -->
  <configuration type="group" cluster="ExampleCluster"
    id="core_example_agents">
    <configuration type="factory" cluster="ExampleCluster"
      pid="net.powermatcher.core.agent.example.ExampleAgent1" id="exampleagent1">
      <property name="id" value="exampleagent1" type="String"></property>
      <property name="bid.power" value="100" type="Double"></property>
      <property name="bid.price" value="0.50" type="Double"></property>
    </configuration>
    <configuration type="factory" cluster="ExampleCluster"
      pid="net.powermatcher.core.agent.example.ExampleAgent2" id="exampleagent2">
      <property name="id" value="exampleagent2" type="String"></property>
      <property name="bid.price" value="0.50" type="Double"></property>
      <property name="bid.power" value="100" type="Double"></property>
      <property name="example.setting" value="Example!" type="String"></property>
    </configuration>
  </configuration>
</configuration>
</nodeconfig>
```

2.6.4 Configuration and Property elements

The Configuration Specification entity is represented in the XML structure by the **configuration** element. The element has an attribute 'pid' that will identify the component type (i.e. it will contain the component name).

The following example shows an configuration example of a PowerMatcher auctioneer component (component name is 'net.powermatcher.core.agent.auctioneer.Auctioneer'):

```
<configuration type="factory" cluster="ExampleCluster"
  pid="net.powermatcher.core.agent.auctioneer.Auctioneer"
  id="auctioneer">
</configuration>
```

The attributes 'id' and 'cluster' identify the Configuration Specification entity and have no relation to the configuration data for the component configuration record. The latter is specified by the **property** elements which are placed as child elements of the configuration element. The 'name' and 'type' attributes of the property element should be equal to the property name and type of the component configuration record respectively.

The configuration element has a 'type' attribute which can have the values 'singleton', 'factory' and 'group'. In the example above it is defined that the Auctioneer component is a factory component, which can have multiple instances within an OSGi container. The Configuration Admin will create a configuration record using the value in the PID as a factory PID. It will generate a unique id, consisting of the component name and a unique value, and assign it to the 'pid' field of the configuration record. In case it would have been a singleton component, the PID of the configuration record would have been equal to the component name.

A 'group' type configuration can logically group other configurations together. However it is also a means of defining a set of common properties for all child configurations within that group. If the properties are placed at the root configuration group they are similar to 'global' properties in a programming language. For example, the property "enabled" of the Auctioneer example configuration above could have been defined at

the root. This would have the effect that all PowerMatcher components are enabled by default. In case for a specific component this is not desired, the property can simply be overridden by redefining the property in the child configuration.

A configuration can act as a *template* configuration when the optional attribute 'template' is set to true. In case one of the child configurations of a group configuration is a template, then all child configurations of with the same PID as the template inherit its properties. A template will only be applied to a configuration when the template is a child of the configuration's ancestor and not when it is a grand child.

For example in a PowerMatcher configuration different PowerMatcher clusters may require a connection adapter factory for MQTT, each having one or more of the same attributes. Then it may be sensible to define a template (note the optional attribute *template* is now set, the default is false):

```
<configuration type="factory" template="true" cluster="Root"
  pid="net.powermatcher.core.messaging.mqttv3.Mqttv3ConnectionFactory" id="mqttv3_connection_template">
  <property name="id" value="mqttv3ConnectionFactory" type="String"></property>
  <property name="reconnect.interval" value="10" type="Integer"></property>
  <property name="notification.enabled" value="false" type="Boolean"></property>
  <property name="broker.uri" value="tcp://localhost:1883"
    type="String"></property>
</configuration>
```

If you create configurations whose ancestor is the parent of the template configuration you only need the following definition to create an MQTT connection adapter factory for a specific PowerMatcher cluster:

```
<configuration type="factory" template="false" cluster="ExampleCluster"
  pid="net.powermatcher.core.messaging.mqttv3.Mqttv3ConnectionFactory" id="mqttv3ConnectionFactory">
</configuration>
```

All other properties will be inherited from the template and need not to be defined, unless you want override the default value from the template.

2.6.5 Node Configuration

A PowerMatcher runtime environment is called a *node* and is identified by a node id. The Node Configuration is represented by the nodeconfig element in the XML structure. The configurations that make up the node configuration definition are placed as child elements within the nodeconfig element.

The XML excerpt below shows an example Node Configuration (element nodeconfig).

```
<?xml version="1.0" encoding="UTF-8"?>
<nodeconfig id="example-node" name="Example node"
  description="PowerMatcher example node" date="2012-08-29 16:02:06">

  <configuration type="group" id="root">

    ...

  </configuration>
</nodeconfig>
```

The id attribute of nodeconfig contains the id that uniquely identifies the node. The name and description attributes provide additional information and the date field contains the date when the node configuration XML was created.

3. PowerMatcher stand-alone application configuration

3.1 Configuration using property files

A PowerMatcher application can also run as a stand-alone application. When running it without the support of the Pax-Runner OSGi framework you will not have the flexibility of updating the configuration dynamically or benefit from the provisioning function provided by the Apache ACE module. However, the setup and the configuration is very straight forward and simple using a Java properties file.

Running a PowerMatcher application as a stand-alone application (Java main) requires a configuration stored in a Java properties file. The properties file should be named 'agent_config.properties' and should be located in the same directory as the directory from where the application is started.

A stand-alone application will not require the configuration of the adapters as is required when running it as an OSGi application. The program that reads the property file and starts the components automatically adds the required adapters. It will also start a broker manager.

3.2 Configuring adapter factories

In the agent_config.properties file adapter factories are configured using the 'adapter.factory.' prefix in the property name before the id of the factory. The syntax for a property setting for an adapter factory is:

```
adapter.factory.<factory_id>.<factory_property>
```

The type of a factory is defined by the 'class' property name. In the following example the configuration properties specify the configuration of a MarketBasisAdapterFactory with id 'marketbasis':

```
# Define adapter factories for direct connections, without using message broker
adapter.factory.marketbasis.class=net.powermatcher.core.agent.marketbasis.adapter.MarketBasisAdapterFactory
adapter.factory.directprotocol.class=net.powermatcher.core.direct.protocol.adapter.DirectProtocolAdapterFactory
adapter.factory.directlogging.class=net.powermatcher.core.direct.protocol.adapter.DirectLoggingAdapterFactory
```

The properties that define the parameters of the adapter, like for example currency, commodity and price range for the market basis, can be specified as adapter factory properties, or as properties of the component that uses the adapter (the auctioneer in case of the market basis). In this example these properties are specified with the configuration of the auctioneer, in the following section.

3.3 Configuring components

In the agent_config.properties file the individual components are configured using the 'agent.' prefix in the property name before the id of the component. The syntax for a property setting for a component is:

```
agent.<component_id>.<component_property>
```

The type of a component is defined by the 'class' property name. In the following example the configuration properties specify the configuration of a Concentrator with id 'concentrator1' which is connected to the Auctioneer with id 'auctioneer1', as well as to the objective agent with id 'objectiveagent1'.

```
# Auctioneer1 configuration
agent.auctioneer1.class=net.powermatcher.core.agent.auctioneer.Auctioneer
agent.auctioneer1.id=auctioneer1
agent.auctioneer1.enabled=true
agent.auctioneer1.agent.adapter.factory=marketbasis
# Price in currency, to define the market basis
agent.auctioneer1.commodity=electricity
agent.auctioneer1.currency=EUR
agent.auctioneer1.minimum.price=0.00
agent.auctioneer1.maximum.price=0.99
agent.auctioneer1.price.steps=100
agent.auctioneer1.significance=2
agent.auctioneer1.market.ref=0
#agent.auctioneer1.matcher.agent.bid.log.level=FULL_LOGGING
agent.auctioneer1.matcher.aggregated.bid.log.level=FULL_LOGGING
#agent.auctioneer1.matcher.price.log.level=FULL_LOGGING

# Concentrator1 configuration
```

```
agent.concentrator1.class=net.powermatcher.core.agent.concentrator.Concentrator
agent.concentrator1.id=concentrator1
agent.concentrator1.enabled=true
# The concentrator has 2 direct agent adapters, one to the auctioneer, one to the objective agent.
agent.concentrator1.agent.adapter.factory=directprotocol,directprotocol
agent.concentrator1.matcher.id=auctioneer1,objectiveagent1
```

The PowerMatcher framework will instantiate a component of the type specified in the class property and assign the remaining properties.

The 'matcher.id' property is a property for the protocol adapter which is defined at the level of the concentrator. Each adapter that is created for a component automatically inherits the configuration properties from the component. The configuration properties of an adapter factory specify the default values for properties that are not defined at the component level. Likewise, the market basis properties that are defined at the auctioneer level are used by the market basis adapter that connects to the auctioneer.

Just as in the OSGi runtime, the required adapters will be created automatically by the framework. In the above example an agent adapter will be created from an adapter factory with the id 'marketbasis' for the auctioneer. For the concentrator two agent adapters will be created from the adapter factory with the id 'directprotocol' (for the concentrator this means that the published bid will be sent to both the auctioneer and the objective agent).

The adapters will get the same id as that of the component by default (i.e. 'concentrator1'), or the id specified in the connector.id property of the component. The latter is a special case that allows a single adapter to be shared between multiple components, like for example a single MQTT connection that is shared between all logging adapters.

It is also possible to specify global properties by omitting the 'adapter.factory.<factory id>' or 'agent.<component id>' prefix. For example, the same factory id, cluster id, and update interval can be defined for all agents (and factories) specifying the following 'global' properties:

```
# Default scheduler adapter factories
time.adapter.factory=time
scheduler.adapter.factory=scheduler

# Global properties.
cluster.id=ExampleCluster
# Change from default 300 seconds to 10 seconds
update.interval=10
# By default logging will go to the configured CSVLoggingAgent agent.
log.listener.id=csvlogging
```

3.4 Running the stand-alone application

The stand-alone application (see paragraph 5.12.1 *Build the stand-alone Core application*) can be started from the directory where it is deployed using the following command from the command shell prompt:

```
java -jar pwmcoredemoapp.jar
```

In case you want to specify another or an additional configuration file you can specify:

```
java -jar pwmcoredemoapp.jar -agentConfig my_project_agent_config.properties
```

4. Creating a PowerMatcher configuration

A basic Power Matcher runtime environment consists of the components that are delivered with the PowerMatcher Core distribution. That is an Auctioneer, a Concentrator, and optionally an Objective Agent and a CSV Logging agent. One also needs at least one agent that implements the PowerMatcher protocol which sends regular bid updates. For our basic configuration we will use the Test Agent that is included with PowerMatcher Core. This section describes how to create the simplest possible configuration for such a runtime setup, using direct connections (procedure calls).

Messaging connections instead of direct connections requires a slightly more involved configuration that sets up an agent protocol adapter factory, message protocol adapter factory and an MQTT connection factory. This is not covered in this chapter.

The PowerMatcher framework will activate a component when the required dependencies are fulfilled. For example, an Auctioneer requires next to an Auctioneer configuration also at least a Market Basis Adapter that defines the market parameters.

The following table lists the components and the required component adapters for which a configuration has to be created in order to activate the component:

Component	Main component	Adapter
Auctioneer	PowerMatcher Auctioneer	Market Basis adapter
		Direct Logging adapter, to csv logging agent
Concentrator	PowerMatcher Concentrator	Direct Protocol adapter, to auctioneer and objective agent
		Direct Logging adapter, to csv logging agent
Objective Agent	PowerMatcher Objective Agent	Direct Protocol adapter, to auctioneer
		Direct Logging adapter, to csv logging agent
CSV Logging Agent	PowerMatcher CSV Logging Agent	None
Test Agent	PowerMatcher Test Agent	Direct Protocol adapter, to concentrator
		Direct Logging adapter, to csv logging agent
all		Time and scheduler adapter

Table 1 The components and component adapters required for a basic PowerMatcher application.

In this chapter we will illustrate the creation of a basic PowerMatcher runtime using the following alternate approaches:

- Create a configuration manually using the OSGi web console.
- Configuration of an OSGi runtime using an XML configuration file.
- Configuration of a standard Java runtime using a properties file.

4.1 OSGi configuration example

4.1.1 Manual configuration using the OSGi web console

Using the Apache Felix Web Console the configuration can be created manually. Each configuration object for a component that is created with the web console is already prefilled with the default values, which makes a quick setup very easy.

Open the Apache Felix Web Console (<http://localhost:8080/system/console>). Then navigate to the Configuration tab and find the configuration name as listed in the table. Click the '+' to create a new configuration instance or edit a single configuration instance (e.g. for the Auctioneer) and fill in the properties as specified in the table. Select the Save button to create the configuration.

Component	Required OSGi configurations (name/class name)	Set property
Time Adapter	PowerMatcher Time Adapter Factory / net.powermatcher.core.scheduler.TimeAdapterFactory	Accept all default values
Scheduler Adapter	PowerMatcher Scheduler Adapter Factory / net.powermatcher.core.scheduler.SchedulerAdapterFactory	Accept all default values

Protocol Adapter	PowerMatcher Direct Protocol Adapter / net.powermatcher.core.direct.protocol.adapter.component. DirectProtocolAdapterFactory	<i>Accept all default values</i>
Logging Adapter	PowerMatcher Direct Logging Adapter / net.powermatcher.core.direct.protocol.adapter.component. DirectLoggingAdapterFactory	<i>Accept all default values</i>
Market Basis Adapter	PowerMatcher Market Basis Adapter / net.powermatcher.core.agent.marketbasis.adapter. MarketBasisAdapterFactory	<i>Accept all default values</i>
Auctioneer	PowerMatcher Auctioneer / net.powermatcher.core.agent.auctioneer.Auctioneer	Id: auctioneer
Concentrator	PowerMatcher Concentrator / net.powermatcher.core.agent.concentrator.Concentrator	Id: concentrator Matcher id: auctioneer
Objective Agent	PowerMatcher Objective Agent / net.powermatcher.core.agent.objective.ObjectiveAgent	Id: objectiveagent Matcher id: auctioneer Objective bid: (64,0);(64,-1000.0)
CSV Logging Agent	PowerMatcher CSV logging Agent / net.powermatcher.core.agent.logging.CSVLoggingAgent	Id: csvlogging
Test Agent	PowerMatcher Test Agent / net.powermatcher.core.agent.test. TestAgent	Id: testagent1 Matcher id: concentrator

Table 2 Component properties to be defined in the web console for a minimal configuration.

For the sake of simplicity, this configuration uses default values for almost all properties. This includes the cluster ID, the logging agent identifier and logging level, and the adapter factory identifiers.

The following screenshot shows the creation of the PowerMatcher Auctioneer configuration in the Web Console:

The screenshot displays the Apache Felix Web Console interface for configuring the 'PowerMatcher Auctioneer' component. The configuration is as follows:

- Cluster id:** DefaultCluster
- Cluster ID (cluster.id):** (empty field)
- Id:** auctioneer
- Component ID (id):** (empty field)
- Enabled:** ☒ (Component enabled (enabled))
- Time adapter factory:** timeAdapterFactory
- Scheduler adapter factory:** schedulerAdapterFactory
- Agent adapter factory:** marketBasisAdapterFactory
- Matcher adapter factory:** (empty field)
- Pricing adapter factory:** (empty field)
- Logging adapter factory:** directLoggingAdapter
- Log listener id:** csvlogging
- Update interval:** 30
- Bid expiration time:** 300
- Matcher agent bid log level:** Full logging
- Matcher aggregated bid log level:** No logging
- Matcher price log level:** Full logging

Configuration Information:

- Persistent Identity (PID): [Temporary PID replaced by real PID upon save]
- Factory Persistent Identifier (Factory PID): net.powermatcher.core.agent.auctioneer.Auctioneer
- Configuration Binding: Unbound or new configuration

Buttons at the bottom: Save, Reset, Abort.

Figure 4 Creating an Auctioneer configuration record in the Apache Web Console.

After configuration you can verify that a component is activated in the 'Components' tab of the Felix Web Console. All components and component adapters that you have configured should have the state 'Active'.

Note that Active means that OSGi has created and activated components. It does not necessary mean that the component is also active at the PowerMatcher level. An Auctioneer, for example, is not started until it has received a market basis from the market basis adapter. The state of PowerMatcher components can be inspected by reviewing the OSGi console output, or by inspecting the events that are logged by the logging agent.

4.1.2 Configuration in the OSGi runtime using XML

The procedure that was followed in the previous section using the Apache Felix Web Console can also be captured into one XML file. In this section we will create an XML that will define the configurations for the same list of components and component adapters in Table 1. The XML configuration will contain the minimum of properties that need to be set. The properties that need to be set are shown in Table 3. In this table common adapter factory properties have been omitted; these are defined as global properties in the XML on the next page.

First we create XML header and the node configuration element with the name 'sample_node':

```
<?xml version="1.0" encoding="UTF-8"?>
<nodeconfig id="sample_node" name="Sample Core node"
  description="Example basic PWM Core configuration using XML." date="2012-06-01">
</nodeconfig>
```

Then we add for each component listed in the table a configuration specification element as a child of the node element. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<nodeconfig id="sample_node" name="Sample Core node"
  description="Example basic PWM Core configuration using XML." date="2012-06-01">
  <configuration type="factory" cluster="DemoPwmCore"
    pid="net.powermatcher.core.agent.auctioneer.Auctioneer" id="auctioneer1">
    <property name="id" value="auctioneer" type="String"></property>
  </configuration>
</nodeconfig>
```

Component	Required OSGi configurations (name/class name)	Set property
Time Adapter	PowerMatcher Time Adapter Factory / net.powermatcher.core.scheduler.TimeAdapterFactory	id: timeAdapterFactory
Scheduler Adapter	PowerMatcher Scheduler Adapter Factory / net.powermatcher.core.scheduler.SchedulerAdapterFactory	id: schedulerAdapterFactory
Protocol Adapter	PowerMatcher Direct Protocol Adapter / net.powermatcher.core.direct.protocol.adapter.component. DirectProtocolAdapterFactory	id: directProtocolAdapterFactory
Logging Adapter	PowerMatcher Direct Logging Adapter / net.powermatcher.core.direct.protocol.adapter.component. DirectLoggingAdapterFactory	id: directLoggingAdapterFactory
Market Basis Adapter	PowerMatcher Market Basis Adapter / net.powermatcher.core.agent.marketbasis.adapter. MarketBasisAdapterFactory	id: marketBasisAdapterFactory
Auctioneer	PowerMatcher Auctioneer / net.powermatcher.core.agent.auctioneer.Auctioneer	id: auctioneer
Concentrator	PowerMatcher Concentrator / net.powermatcher.core.agent.concentrator.Concentrator	id: concentrator matcher.id: auctioneer
Objective Agent	PowerMatcher Objective Agent / net.powermatcher.core.agent.objective.ObjectiveAgent	id: objectiveagent matcher.id: auctioneer objective.bid: (64,0);(64,-1000.0)
CSV Logging Agent	PowerMatcher CSV logging Agent / net.powermatcher.core.agent.logging.CSVLoggingAgent	id: csvlogging
Test Agent	PowerMatcher Test Agent / net.powermatcher.core.agent.test.TestAgent	id: testagent1 matcher.id: concentrator

Table 3 Component properties to be defined in XML for a basic PowerMatcher Core runtime.

In the following example all component and adapter factory configurations of Table 3 have been included in the XML. Note that factory ids must be specified explicitly and have been defined as global properties for the commonly used factories.

```
<?xml version="1.0" encoding="UTF-8"?>
<nodeconfig id="sample_node" name="Sample Core node"
  description="Example basic PWM Core configuration using XML." date="2012-06-01">
  <property name="time.adapter.factory" value="timeAdapterFactory" type="String"/>
  <property name="scheduler.adapter.factory" value="schedulerAdapterFactory" type="String"/>
  <property name="agent.adapter.factory" value="directProtocolAdapterFactory" type="String"/>
  <property name="logging.adapter.factory" value="directLoggingAdapterFactory" type="String"/>
  <property name="log.listener.id" value="csvlogging" type="String"/>
  <configuration type="factory" cluster="DemoPwmCore"
    pid="net.powermatcher.core.agent.auctioneer.Auctioneer" id="auctioneer">
    <property name="id" value="auctioneer" type="String"></property>
    <property name="agent.adapter.factory" value="marketBasisAdapterFactory" type="String"/>
  </configuration>
  <configuration type="factory" cluster="DemoPwmCore"
    pid="net.powermatcher.core.agent.concentrator.Concentrator"
    id="concentrator">
    <property name="id" value="concentrator" type="String"></property>
    <property name="matcher.id" value="auctioneer" type="String"></property>
  </configuration>
  <configuration type="factory" cluster="DemoPwmCore"
    pid="net.powermatcher.core.agent.objective.ObjectiveAgent"
    id="concentrator">
    <property name="id" value="objectiveagent" type="String"></property>
    <property name="matcher.id" value="objectiveagent" type="String"></property>
    <property name="objective.bid" value="(64,0);(64,-1000.0)" type="String"></property>
  </configuration>
  <configuration type="factory" cluster="DemoPwmCore"
    pid="net.powermatcher.core.agent.logging.CSVLoggingAgent"
    id="csvlogging">
    <property name="id" value="csvlogging" type="String"></property>
  </configuration>
  <configuration type="factory" cluster="DemoPwmCore"
    pid="net.powermatcher.core.agent.test.TestAgent"
    id="testagent1">
    <property name="id" value="testagent1" type="String"></property>
    <property name="matcher.id" value="concentrator" type="String"></property>
  </configuration>
  <configuration type="factory" cluster="DemoPwmCore"
    pid="net.powermatcher.core.scheduler.TimeAdapterFactory"
    id="timeAdapter">
    <property name="id" value="timeAdapterFactory" type="String"></property>
  </configuration>
  <configuration type="factory" cluster="DemoPwmCore"
    pid="net.powermatcher.core.scheduler.SchedulerAdapterFactory"
    id="schedulerAdapter">
    <property name="id" value="schedulerAdapterFactory" type="String"></property>
  </configuration>
  <configuration type="factory" cluster="DemoPwmCore"
    pid="net.powermatcher.core.direct.protocol.adapter.component.DirectProtocolAdapterFactory"
    id="directProtocolAdapter">
    <property name="id" value="directProtocolAdapterFactory" type="String"></property>
  </configuration>
  <configuration type="factory" cluster="DemoPwmCore"
    pid="net.powermatcher.core.direct.protocol.adapter.component.DirectLoggingAdapterFactory"
    id="directLoggingAdapter">
    <property name="id" value="directLoggingAdapterFactory" type="String"></property>
  </configuration>
  <configuration type="factory" cluster="DemoPwmCore"
    pid="net.powermatcher.core.agent.marketbasis.adapter.MarketBasisAdapterFactory"
    id="marketBasisAdapter">
    <property name="id" value="marketBasisAdapterFactory" type="String"></property>
  </configuration>
</nodeconfig>
```

4.2 Configuration of a stand-alone Java runtime with a property file

In order to create a stand-alone Java PowerMatcher environment similar to the environments created using the Apache Felix Web console and the XML configuration for the OSGi runtime, the same definitions as listed in Table 3 should be added to a properties file.

By default, PowerMatcher expects the properties file name 'agent_config.properties' in the same directory as the directory from where the application is started. It is possible to specify an alternative properties file path as described in section 3.4. The properties from Table 3 can be specified as follows in a properties file:

```
# Global properties.
cluster.id=DemoPwmCore
# By default logging will go to the configured CSVLoggingAgent agent.
log.listener.id=csvlogging

# Define time & scheduler adapter factories
adapter.factory.timeAdapterFactory.class=net.powermatcher.core.scheduler.TimeAdapterFactory
adapter.factory.schedulerAdapterFactory.class=net.powermatcher.core.scheduler.SchedulerAdapterFactory

# Define adapter factories for direct connections, without using message broker
adapter.factory.marketBasisAdapterFactory.class=net.powermatcher.core.agent.marketbasis.adapter.MarketBasisAdapterFactory
adapter.factory.directProtocolAdapterFactory.class=net.powermatcher.core.direct.protocol.adapter.DirectProtocolAdapterFactory
adapter.factory.directLoggingAdapterFactory.class=net.powermatcher.core.direct.protocol.adapter.DirectLoggingAdapterFactory

# Default scheduler adapter factories
time.adapter.factory=timeAdapterFactory
scheduler.adapter.factory=schedulerAdapterFactory

# Default adapter factories for direct connections, without using message broker
agent.adapter.factory=directProtocolAdapterFactory
logging.adapter.factory=directLoggingAdapterFactory

# Auctioneer properties
agent.auctioneer.class=net.powermatcher.core.agent.auctioneer.Auctioneer
agent.auctioneer.id=auctioneer
agent.auctioneer1.agent.adapter.factory=marketBasisAdapterFactory

# Concentrator properties
agent.concentrator.class=net.powermatcher.core.agent.concentrator.Concentrator
agent.concentrator.id=concentrator
agent.concentrator.matcher.id=auctioneer

# ObjectiveAgent properties
agent.objectiveagent.class=net.powermatcher.core.agent.objective.ObjectiveAgent
agent.objectiveagent.id=objectiveagent
agent.objectiveagent.matcher.id=auctioneer
agent.objectiveagent.objective.bid=(64,0);(64,-1000.0)

# CSVLoggingAgent properties
agent.csvlogging.class=net.powermatcher.core.agent.logging.CSVLoggingAgent
agent.csvlogging.id=csvlogging

# Test Agent 1
agent.testagent1.class=net.powermatcher.core.agent.test.TestAgent
agent.testagent1.id=testagent1
agent.testagent1.matcher.id=concentrator
```

5. Development Environment Setup

PowerMatcher has been developed in the Java programming language. For development of PowerMatcher and PowerMatcher applications the Eclipse Integrated Development Environment is used. This is an open source development environment which is available from the www.eclipse.org web site. Eclipse is extended with the Bndtools 2.0 plug-ins from bndtools.org to facilitate the development of OSGi components. The Bndtools plug-ins are also available under an open source license.

5.1 PowerMatcher maintenance versus application development

There are two different configurations for the PowerMatcher Core development environment.

1. **The development environment for the Core agent and application developer.**
This environment contains the Core Example template source code projects and uses the binary distribution of the PowerMatcher Core libraries and bundles.
2. **The development environment for the PowerMatcher Core developer**, for maintaining and extending the PowerMatcher Core code base.
This environment uses all Core and Core Example source code projects as well as the binary distribution of the PowerMatcher Core libraries and bundles.

This chapter describes how to install the development environment with the part or all of the source code, how to build and develop PowerMatcher Core based applications and how to deploy them.

5.2 Install a Java JDK

Bndtools provides two ways to build OSGi bundles for deployment and release. Since Bndtools 2.0 the built-in Release Bundles wizard can be used to build and release a single or multiple bundles in one step. Alternatively, the Bndtools Ant build scripts can be used as explained in the following sections. However, this is much more involved.

A Java SDK will be required for building the PowerMatcher libraries and bundles using the Bndtools Ant build scripts. Download a Java Standard Edition 6 or 7 SDK from the Internet and install it. For the built-in Bundle Release wizard a Java JRE suffices.

5.3 Eclipse download and install and setup

Download a copy of the Eclipse IDE for Java Developers from the Eclipse site [1] and install it on your workstation. Note: Use the latest 4.1 or 3.7 release.

Start Eclipse. When you start Eclipse you will be prompted for the workspace location. Specify the path of the workspace where you will install later the PowerMatcher Core and/or Core Example sources:

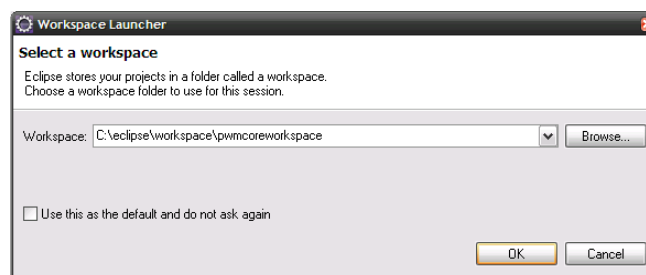


Figure 5 The Eclipse dialog for selecting an existing or new workspace.

Click OK to open the workspace. By default the Java perspective will open showing a window similar to the following figure:

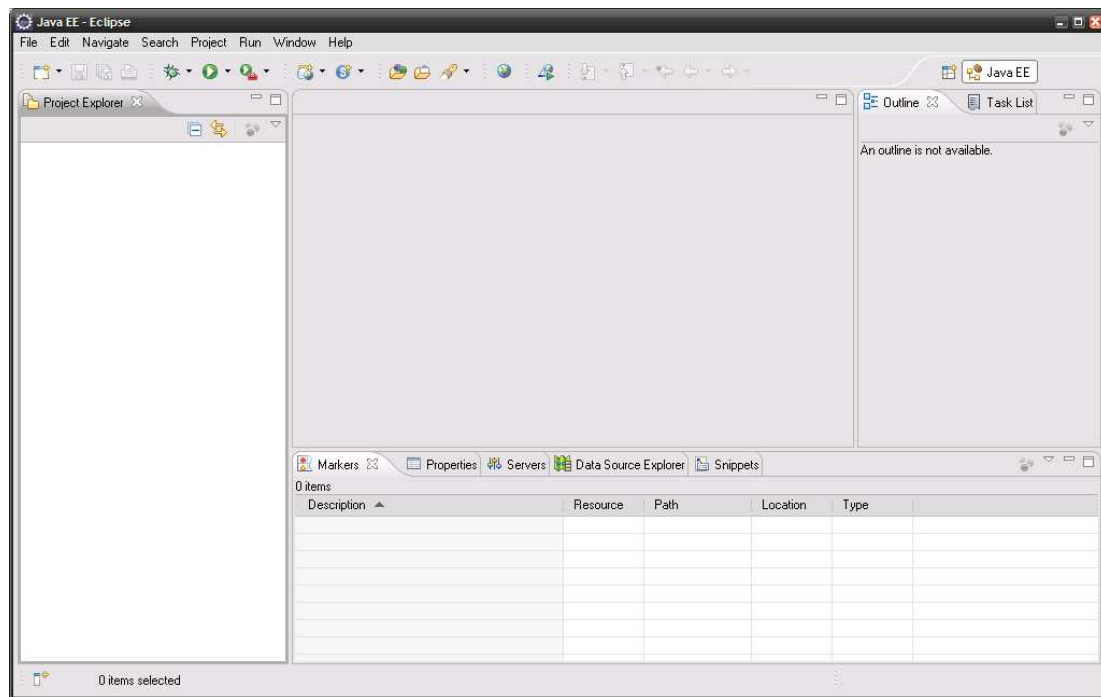


Figure 6 The Eclipse workbench after creating a new workspace.

5.4 Installing the Bndtools plug-in

The Bndtools 2.0 plug-ins can be installed from the Eclipse Marketplace or from the Eclipse download site for Bndtools. Open the Eclipse Marketplace from the Help menu in Eclipse. In the Marketplace window search for 'bnd':

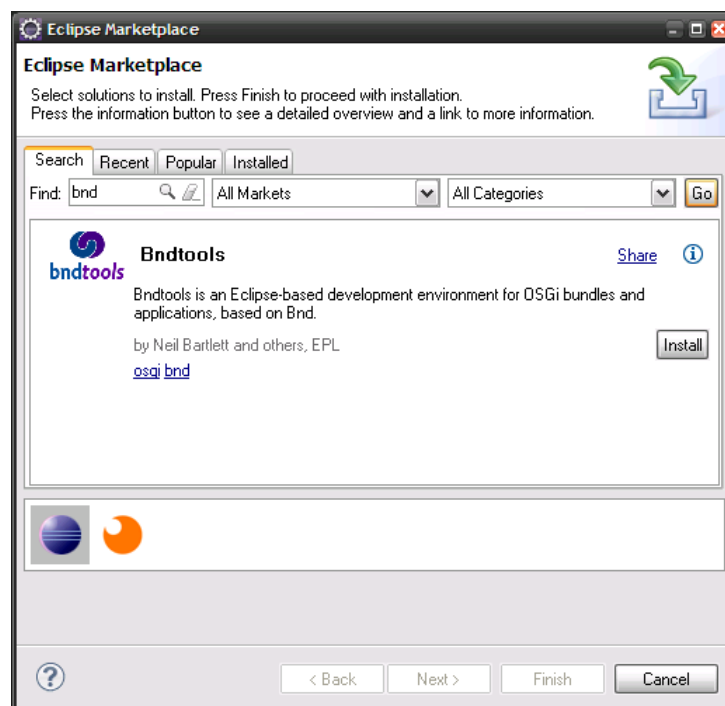


Figure 7 Installing the Bndtools plugin from the Eclipse Marketplace.

Install the Bndtools plug-ins by clicking the Install button. You will be asked to confirm the installation of the requested features and to agree with the license terms. There will be also a security warning because it contains 'some unsigned content'. Ignore this warning and continue the installation. Finally you will be

prompted to restart the workbench to begin to work with Bndtools. After restart Eclipse is ready to work with source code that uses Bndtools artefacts like PowerMatcher Core.

5.5 MQTT broker installation

When using messaging connections instead of direct connections, an MQTT messaging broker is required. Though there are different MQTT broker implementations available, in this document the open source (BSD Licensed) MQTT broker named 'Mosquitto' is briefly described.

Mosquitto implements version 3 of the MQTT protocol. Builds are available for Windows and for different Linux distributions[11]. When you install the version for Microsoft Windows you can run it as a service or directly by invoking the executable in the install directory. The installation is very simple and is guided by an installation wizard.

In the Mosquitto installation directory you can find a configuration file named 'Mosquitto.conf'. In this file you can customize various settings, for example change the default port from 1883 to another port number. If you decide to change this port number remember that you also need to define this port number in the MQTT connection adapter configuration.

5.6 Retrieving the source code

The source code of PowerMatcher Core and Core Example working sets are delivered as an Eclipse project archive compressed in ZIP format.

Download the zip file from the PowerMatcher download site, unzip the file to a directory and remember the location for the next steps.

5.7 IDE Setup

Start Eclipse and create a workspace named, for example, 'pwmcore'.

Change in the Preferences (from the Window menu) the JRE settings. Add the JDK that you installed in the previous step. Select the checkbox to make it the default JRE:

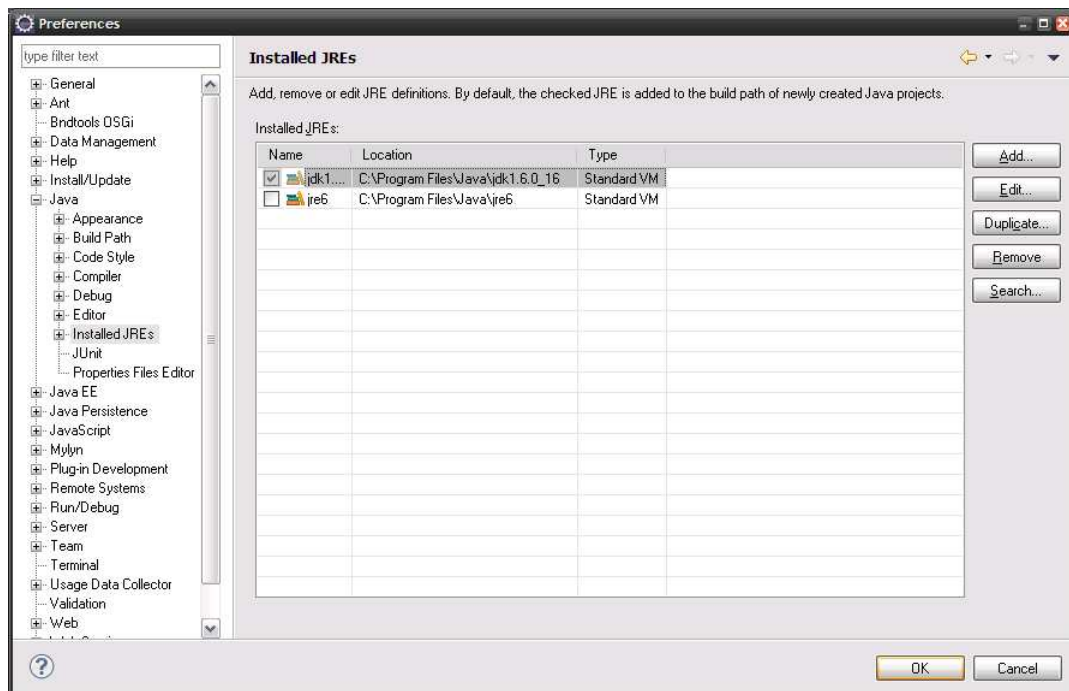


Figure 8 JRE definition and selection in the Eclipse Preferences window.

Open the Bndtools perspective. When you open this perspective for the first time and use any of the Bntools functionality, Bndtools will launch a wizard to create a configuration project named 'cnf' to store its local bundle repository and build settings. Select 'Create a configuration project' and click Next.



Figure 9 Bndtools wizard for creating a configuration project named 'cnf'.

In the next screen select the 'Default Configuration' template and click Finish.

Configuration Template

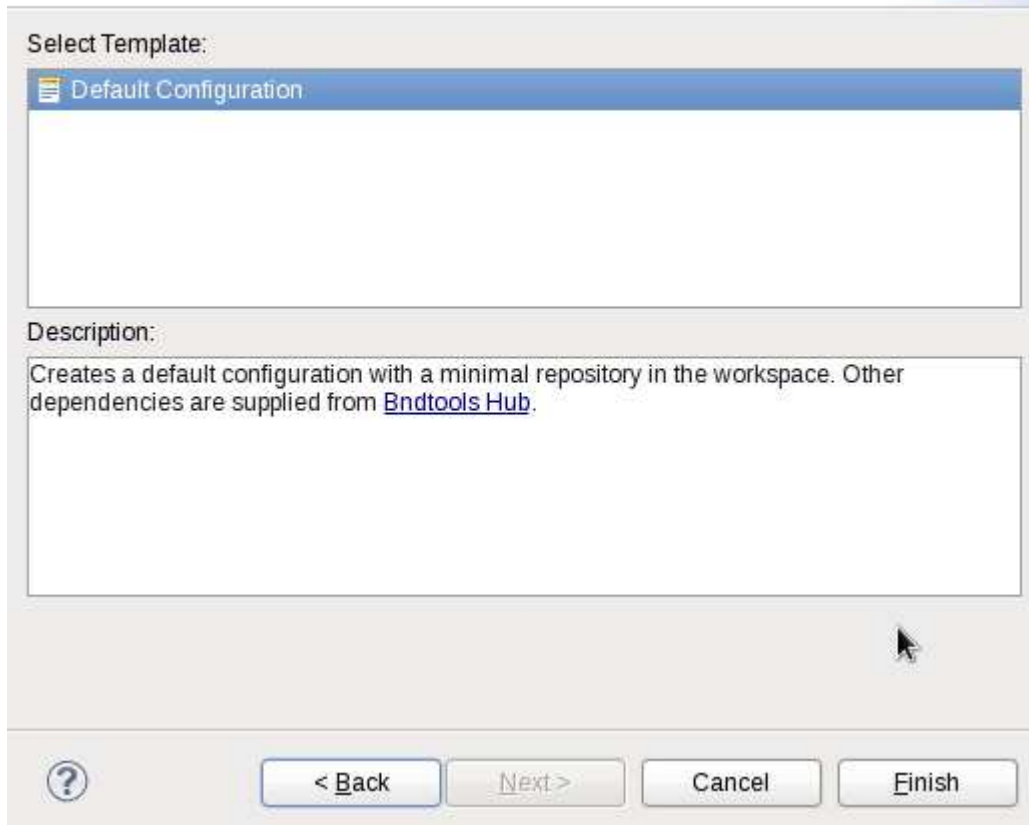


Figure 10 Select the Default Configuration template for the 'cnf' project.

Bndtools will create 'cnf' project and automatically download a set of libraries that are common in developing OSGi components:

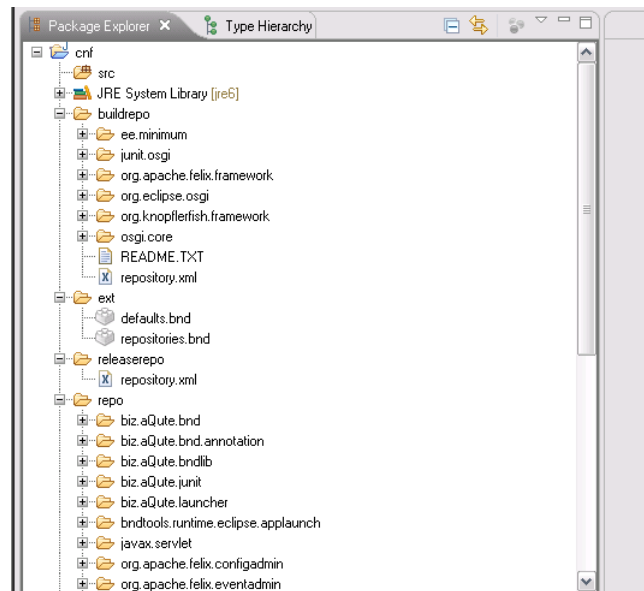


Figure 11 The 'cnf' project created by Bndtools prefilled with common bundles.

Add the following bundle jars, required for compiling and running the PowerMatcher Core and Core Example projects, to the Local repository by dragging and dropping the jars on 'Local' repository in the Bndtools Repositories View.

Function	JAR file	Description	Download
Eclipse Equinox OSGi implementation	org.eclipse.equinox.common.jar	Individual bundles that are prerequisites for MicroBroker in PowerMatcher Extension. For PowerMatcher Core this step can be skipped.	See [3], choose build version then Add-on bundles and find specified jar.
	org.eclipse.equinox.preferences.jar		
Base 64	org.apache.commons.codec	Encoders and decoders such as Base64	See [5].
Logging	slf4j.api-1.7.2.jar	Simple Logging Facade for Java (SLF4J)	See [8].
	ch.qos.logback.classic-1.0.9.jar	Logging framework.	See [9].
	ch.qos.logback.core-1.0.9.jar	Logging framework.	
MQTT client	org.eclipse.paho.client.mqttv3	Eclipse Paho open source implementation of messaging client for the MQTT version 3 protocol.	See [10]

Table 4 Required local libraries for PowerMatcher Core development.

All other prerequisite libraries are available from the Bndtools Hub repository and are downloaded and cached automatically by Bndtools.

5.8 Importing the PowerMatcher Core binary distribution

This step is required for both the agent developer workspace, as well as for the workspace for PowerMatcher maintenance. In this step a new Bndtools local repository is added to the workspace that will contain the PowerMatcher Extension bundle library jars.

1. Create workspace folder /cnf/corerepo
2. Edit /cnf/ext/repositories.bnd and insert the following line to add a new repository labeled 'Core':

```
aQute.bnd.deployer.repository.LocalIndexedRepo; name=Core;
local=${workspace}/cnf/corerepo;pretty=true,\
```
3. Add all PowerMatcher Core bundle jars to the Core repository by dragging and dropping the jars on 'Core' in the Repositories View.

5.9 Importing the PowerMatcher Core and Core Example sources

The projects can now be imported into the workspace. For the agent developer workspace, import only the Core Examples projects into the workspace. For the PowerMatcher developer workspace, import both the Core and the Core Examples projects into the workspace.

First switch off the automatic build option by de-selecting 'Build automatically...' in the Project menu. Then use the Import... wizard and select 'Import existing projects into workspace' and point the wizard to the folder where you exported the project archive file. Select all projects and check 'Copy projects into workspace' and import the projects by clicking the Finish button.

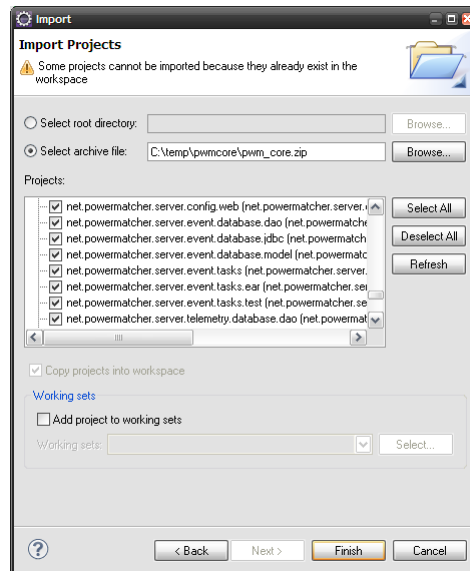


Figure 12 Importing existing projects using the Import Projects wizard of Eclipse.

In order to compile the sources successfully, a user library preferences file has to be imported. This file contains the required user library definitions that are referenced by the PowerMatcher Core projects. A user library preferences file can be imported via the Eclipse Preferences window

The user library file is named 'PowerMatcher Core.userlibraries' and is located in the `net.powermatcher.core.launcher.main.template` project. After the import three user library definitions have been added.

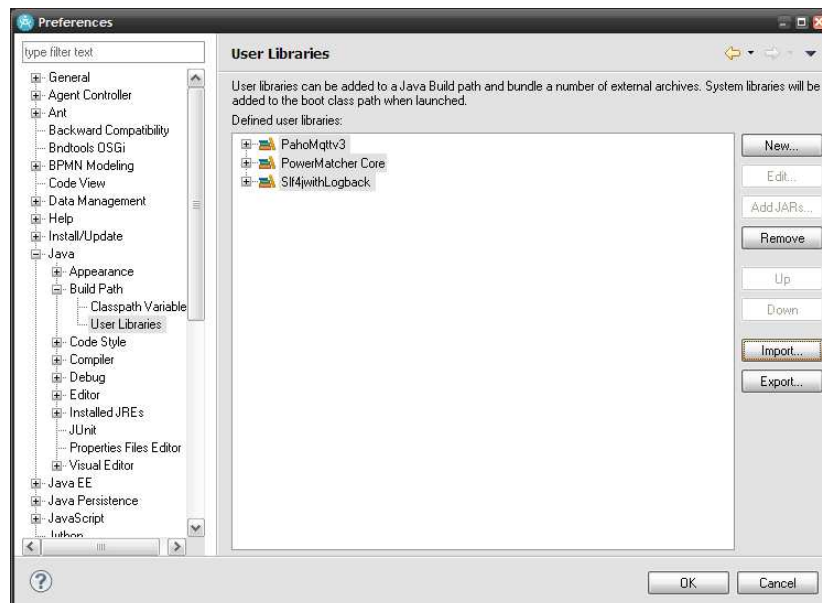


Figure 13 The user library definitions in the Eclipse preferences window.

After the import, restart the workbench (File -> Restart). The cnf repository cache of BND tools will now be cleared. From the message bar at the bottom of the window you can see when this task is running. If the task is completed check the 'Build automatically...' option or hit Ctrl-B (i.e. Build All). All projects should now compile successfully.

You can organize the imported projects using working sets as desired.

Alternatively, if you have access to the PowerMatcher subversion repository, the project and working sets references are defined in the Team Project Set file named 'net.powermatcher-core.psf' in the project

net.powermatcher.core.resource. The projects can be imported by selecting the option 'Import Project Set...' from the context menu.

5.10 Code style and formatting preferences

Common coding standards are enforced by an Eclipse 'style and formatting preferences' file that has been designed for the PowerMatcher development. Apply this style when you create new projects or modify existing ones.

You can import the preferences from:

```
/net.powermatcher.core.resource/powermatcher-eclipse-preferences.epf
```

The style and profile definitions in the epf file can also be viewed in a more readable format in the following two files:

```
/net.powermatcher.core.resource/powermatcher-eclipse-style-profile.xml
```

```
/net.powermatcher.core.resource/powermatcher-eclipse-formatter-profile.xml
```

In Eclipse or Rational Application Developer, open the Import Wizard (File->Import...) and choose in the category 'General' the 'Preferences' item. The Import Preferences Wizard will open. Browse to the .epf file in the net.powermatcher.core.resource project in your workspace and click Finish. The code style and formatting preferences are now active.

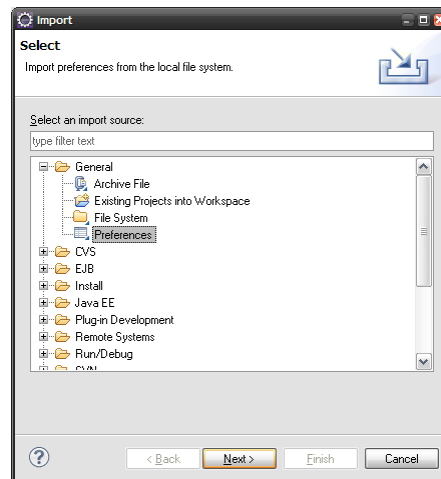


Figure 14 Eclipse preferences import wizard.

The style and formatting preferences can be applied to your code by selecting your projects and choosing Source->Clean up... (via the menu bar or context menu). A Clean Up wizard will popup which will guide you through the reformatting process. You can preview the proposed changes, but it is preferred that you accept all by clicking the Finish button.

5.11 Launching the application from the workbench

This paragraph describes how to run a PowerMatcher application from the Eclipse workspace. There are two options: running as a stand-alone Java application and running in an OSGi runtime.

5.11.1 Run as a stand-alone Java application

A stand-alone application can be simply run from the workbench using the Run icon in the toolbar if you have defined a launch configuration. A sample application and launch configuration has been created in the 'net.powermatcher.core.launcher.main.template' project. If you expand the project you will notice the launch configuration stored as a file named 'PowerMatcher Launch Template.launch'. You can launch the application from the 'Run-as...' option of the context menu or you can select it from the Run button in the Eclipse toolbar (see Figure 15).

The application will use the agent_config.properties file in that same project. Modify it to customize your own PowerMatcher Core application.

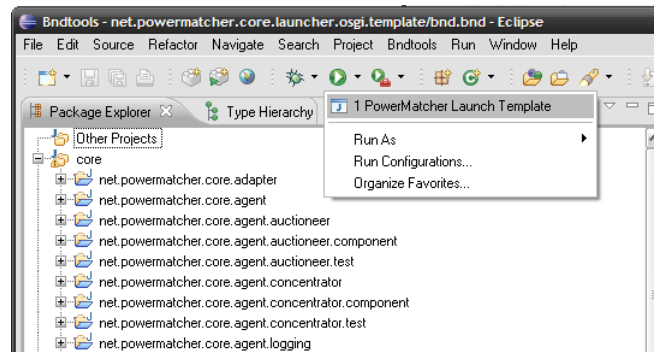


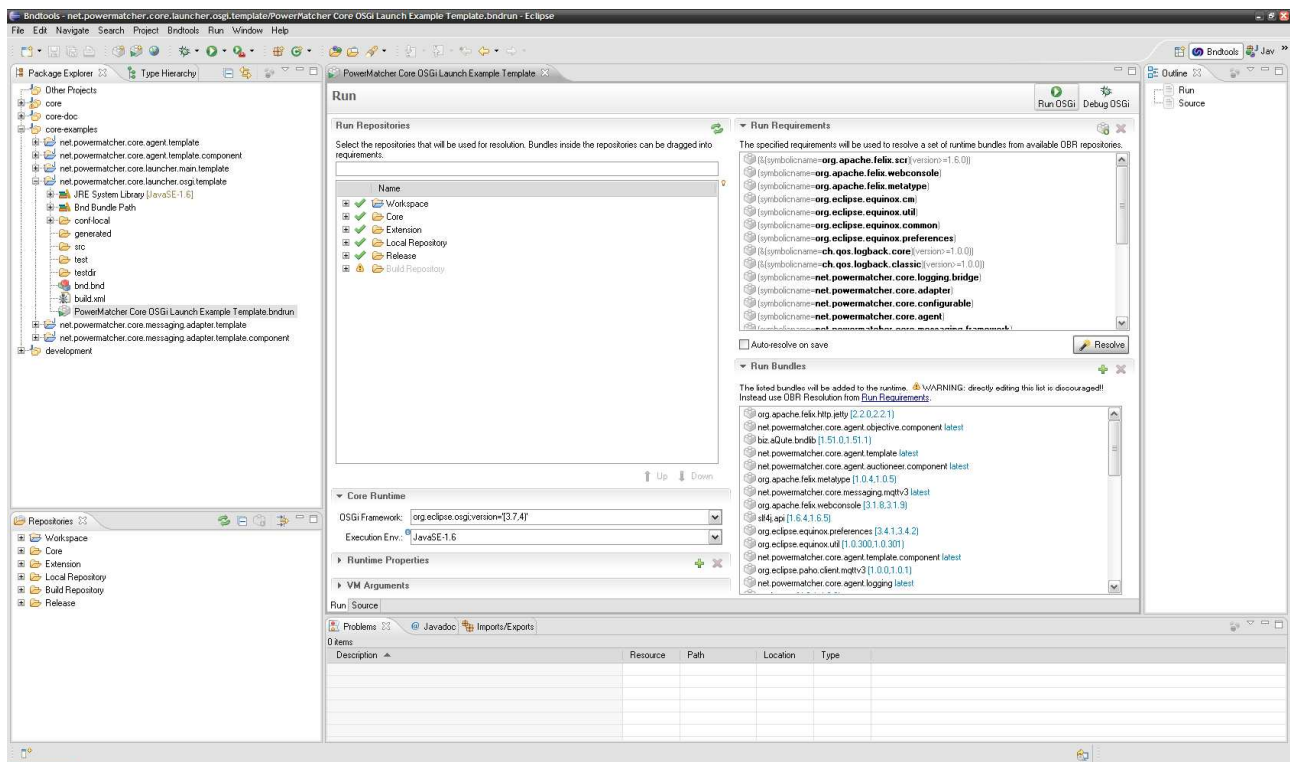
Figure 15 Launching the PowerMatcher application from the Run-button in the Eclipse toolbar.

Note that the stand-alone Java launch configurations will use the PowerMatcher library jars from the Bntools Core repository through the Eclipse User Library definitions imported in section 5.9.

5.11.2 Run as an OSGi application

A PowerMatcher application running in the OSGi runtime environment can be started from a 'bndrun' file. In the project `net.powermatcher.core.launcher.osgi.template` you can find an example run file named 'PowerMatcher Core OSGi Launch Example Template.bndrun'.

The launch configuration will start an Equinox OSGi runtime with some active components since it reads an XML configuration from the `conf-local` directory.



5.12 Building a PowerMatcher Core application

5.12.1 Build the stand-alone Core application

Make sure the source code is compiled by verifying if the option 'Build Automatically' is checked in the Project menu. If not check the option and verify the compilation progress in the status bar at the bottom of the window.

From the File menu select Export... menu option. The Export dialog will appear.

Select the item Java>Runnable JAR and hit next:

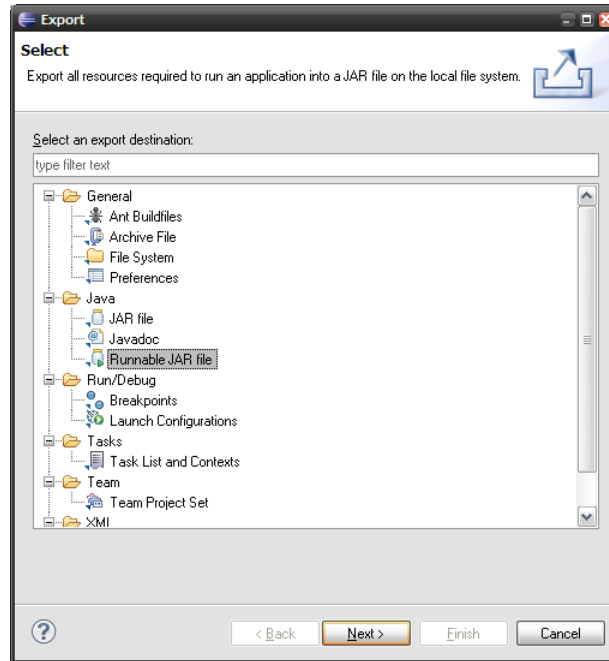


Figure 16 Selecting the 'Runnable JAR file' option from the Eclipse Export wizard.

The page 'Runnable JAR file export' will appear. Complete the fields as in the following figure:

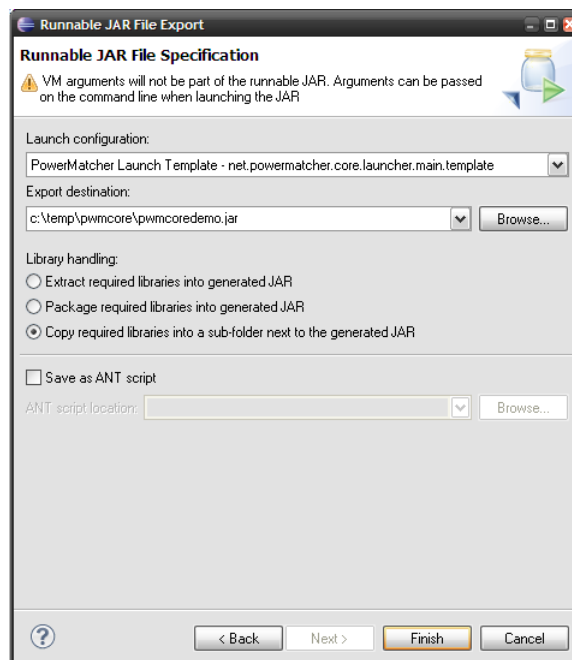


Figure 17 Eclipse wizard for exporting a runnable JAR file.

The wizard will create a runnable jar file named `pwmcoredemo.jar` and create a directory named `pwmcoredemo_lib` that contains the required runtime libraries. A launch configuration is required to generate the runnable JAR file. When creating your own PowerMatcher application you can create a project that contains a similar launcher main class and launch configuration file.

The application needs also the `logback.xml` configuration file for logging and the `agent_config.properties` for the PowerMatcher component configuration. Copy these files from the `net.powermatcher.core.launcher.main.template` project to the directory of the exported JAR file.

5.12.2 Build the PowerMatcher Core OSGi bundle and library jars

If you run a PowerMatcher OSGi application from the workspace, the launch configurations have been set up to use the bundle jars located in the 'generated' directory of the projects. Besides building and releasing individual bundles using the Bndtools Release Bundle wizard on the bnd.bnd file in a bundle project, it is also possible to build all bundles using an Ant script for PowerMatcher Core.

Note that when running a PowerMatcher application as a standard Java application, the library jars from the Bndtools Core repository are referenced through the Eclipse User Library definitions. In this case the jars located in the 'generated' directory of the projects will not be used.

Bndtools provides two ways to build OSGi bundles for deployment and release. Since Bndtools 2.0 the built-in Release Bundles wizard can be used to build and release a single or multiple bundles in one step. Alternatively, the Bndtools Ant build scripts can be used as explained below. However, this is much more involved.

To build the Core bundles using the Release Bundles wizard, select one or more projects, or select the working set containing all projects, and select Release Bundles from the context menu. By default, the wizard releases all bundles to the Release repository, as shown in the following figure.

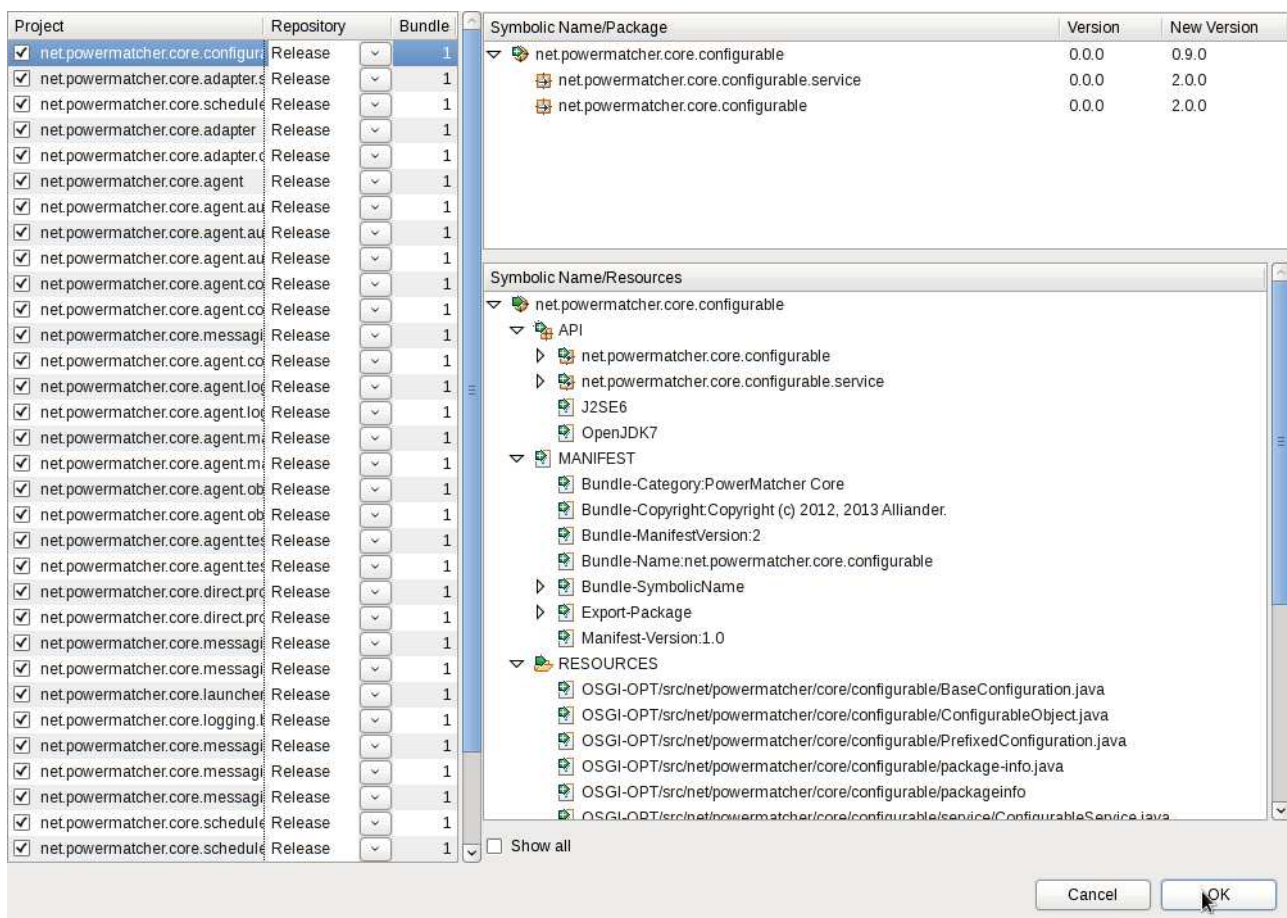


Figure 18 Build Core bundles using the Bndtools Release Bundles wizard.

To build and release the bundles to a different repository, change the `-releaserepo` setting in `/cnf/ext/repositories.bnd`.

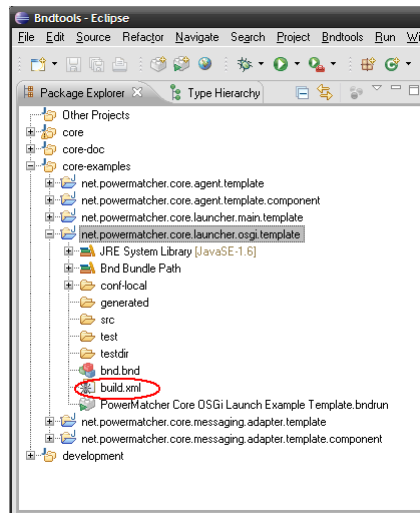


Figure 19 Build Core bundles with the build.xml script in the OSGi example project.

The build and release using Ant is started by selecting the build.xml file and choosing “Run As > ANT Build...”. The default Ant target is ‘build’. To update the bundles in the Bndtools Core repository in the workspace, the ‘release’ target must be selected and the `-releaserepo` setting in `/cnf/ext/repositories.bnd` must be set to Core.

All PowerMatcher Core and Core Example bundles are built through the Ant build.xml file in the `net.powermatcher.core.resource` project. Individual bundles can be built and released via Run As > Ant Build on the build.xml in a bundle project.

Bndtools has also an option to include sources in the generated bundle jars:

`-sources:` true|false

The sources are included under OSGI-OPT in the generated bundle jar. This option is true in the defaults generated by Bndtools in `/cnf/ext/defaults.bnd`. The default can be changed in `/cnf/ext/defaults.bnd` or can be overridden in `/cnf/build.bnd`.

Note that the build and release may take a considerable amount of time. The reason is that for each bundle references, Bndtools rebuilds and releases all prerequisite bundles recursively.

6. Appendix A: PowerMatcher Components

6.1 Core components

This section describes the core components, their configuration properties and the dependencies.

6.1.1 Auctioneer

The auctioneer component creates an auctioneer instance, which will receive all the bids of the other agents as a single bid or as an aggregated bid via one or more concentrators. It is responsible for defining and sending the market basis and calculating the equilibrium based on the bids from the different agents in the topology. This equilibrium is communicated to the agents down the hierarchy in the form of price update messages.

Component: Auctioneer			Type: factory component	
Class: net.powermatcher.core.agent.auctioneer.Auctioneer				
Dependencies: Market Basis Adapter Matcher Adapter (optional) Logging Adapter (optional) Pricing Adapter (optional)				
Property	Type	Req.	Default	Description
id	String	Y	-	Component instance id e.g. 'auctioneer'
cluster.id	String	N	DefaultCluster	PowerMatcher cluster id.
enabled	boolean	N	true	If true component will not be activated when all dependencies are resolved. Otherwise if false it will never be activated.
update.interval	int	N	30	Number of seconds of message interval.
bid.expiration.time	int	N	300	Number of seconds before a received update bid from client agents expires.
time.adapter.factory	String	N	timeAdapterFactory	The identifier of the adapter factory for creating the time service adapter.
scheduler.adapter.factory	String	N	schedulerAdapterFactory	The identifier of the adapter factory for creating the scheduler adapter.
agent.adapter.factory	String	Y	marketBasisAdapterFactory	The identifier of the adapter factory for creating the market basis adapter. The market basis adapter uses the agent connector interface of the matcher to publish market basis updates.
matcher.adapter.factory	String	N		The Identifier of the adapter factory for creating the matcher adapter. Direct connections do not require a matcher adapter, and for this reason the default is an empty string. Fill in the default identifier "matcherProtocolAdapterFactory" of the messaging protocol adapter factory to receive bid updates from and publish price updates to the messaging bus. Multiple adapters can be configured by using a comma-separated list of factory ids.
pricing.adapter.factory	String	N		The identifier of the optional adapter factory for creating an adapter for a custom equilibrium price calculation algorithm. By default the built-in standard algorithm is used.
logging.adapter.factory	String	N	directLoggingAdapterFactory	The identifier of the adapter factory for creating an adapter for logging PowerMatcher bid and price events. By default the adapter that connects directly to the logging agent is used. It is also possible to configure the messaging logging adapter that publishes log messages to the bus. Multiple adapters can be configured by using a comma-separated list of factory ids.
log.listener.id	String	N	csvlogging	The id of the component that will be listening for the logging messages. The direct logging adapter uses this id to locate the component to make the direct connection to. The messaging logging adapter uses this id to address the log listener in the publication topic. When multiple adapters are configured for logging.adapter.factory, multiple ids must be configured here by using a comma-separated list of log listener ids.
matcher.agent.bid.log.level	String	N	FULL_LOGGING	The logging level of PowerMatcher bid updates received by the matcher. This property can be set to NO_LOGGING, PARTIAL_LOGGING or FULL_LOGGING. For partial logging, the full bid curve is omitted and only the minimum, maximum and effective power values are logged.
matcher.aggregated.bid.log_level	String	N	NO_LOGGING	The logging level of PowerMatcher aggregated bid of the matcher. This property can be set to NO_LOGGING, PARTIAL_LOGGING or FULL_LOGGING. For partial logging, the full bid curve is omitted and only the minimum, maximum and effective power

				values are logged.
matcher.price.log.level	String	N	FULL_LOGGING	The logging level of published PowerMatcher price updates. This property can be set to NO_LOGGING or FULL_LOGGING.

6.1.2 Concentrator

The concentrator is a factory component from which several instances can be created. The concentrator receives the bids from the agents and forwards this in an aggregate bid up in the hierarchy to another concentrator or to the auctioneer directly. It will receive price update messages from the auctioneer and forward them to its connected agents.

Component: Concentrator				Type: factory component
Class: net.powermatcher.core.agent.concentrator.Concentrator				
Dependencies: Agent Adapter Matcher Adapter (optional) Logging Adapter (optional)				
Property	Type	Req.	Default	Description
id	String	Y	-	Component instance id e.g. 'concentrator'
cluster.id	String	N	DefaultCluster	PowerMatcher cluster id.
enabled	boolean	N	true	If true component will not be activated when all dependencies are resolved. Otherwise if false it will never be activated.
update.interval	int	N	30	Number of seconds of message interval.
bid.expiration.time	int	N	300	Number of seconds before a received update bid from client agents expires.
time.adapter.factory	String	N	timeAdapterFactory	The identifier of the adapter factory for creating the time service adapter.
scheduler.adapter.factory	String	N	schedulerAdapterFactory	The identifier of the adapter factory for creating the scheduler adapter.
agent.adapter.factory	String	Y	directProtocolAdapterFactory	The identifier of the adapter factory for creating the agent adapter. By default the adapter that connects directly to the matcher is used. Fill in the default identifier "agentProtocolAdapterFactory" of the messaging protocol adapter factory to publish bid updates to and receive price updates from the messaging bus. Multiple adapters can be configured by using a comma-separated list of factory ids.
matcher.id	String	Y	-	The id of the matcher to publish bids to and receive price updates from. The direct protocol adapter uses this id to locate the parent matcher to make the direct connection to. The messaging protocol adapter uses this id to address the parent matcher in the publication topic. When multiple adapters are configured for agent.adapter.factory, multiple ids must be configured here by using a comma-separated list of log listener ids. This can, for example, be used to publish bids to the auctioneer as well as to an objective agent.
matcher.adapter.factory	String	N		The identifier of the adapter factory for creating the matcher adapter. Direct connections do not require a matcher adapter, and for this reason the default is an empty string. Fill in the default identifier "matcherProtocolAdapterFactory" of the messaging protocol adapter factory to receive bid updates from and publish price updates to the messaging bus. Multiple adapters can be configured by using a comma-separated list of factory ids.
logging.adapter.factory	String	N	directLoggingAdapterFactory	The identifier of the adapter factory for creating an adapter for logging PowerMatcher bid and price events. By default the adapter that connects directly to the logging agent is used. It is also possible to configure the messaging logging adapter that publishes log messages to the bus. Multiple adapters can be configured by using a comma-separated list of factory ids.
log.listener.id	String	N	csvlogging	The id of the component that will be listening for the logging messages. The direct logging adapter uses this id to locate the component to make the direct connection to. The messaging logging adapter uses this id to address the log listener in the publication topic. When multiple adapters are configured for logging.adapter.factory, multiple ids must be configured here by using a comma-separated list of log listener ids.
matcher.agent.bid.log.level	String	N	FULL_LOGGING	The logging level of PowerMatcher bid updates received by the matcher. This property can be set to NO_LOGGING, PARTIAL_LOGGING or FULL_LOGGING. For partial logging, the full bid curve is omitted and only the minimum, maximum and effective power values are logged.

matcher.aggregated.bid.log_level	String	N	NO_LOGGING	The logging level of PowerMatcher aggregated bid of the matcher. This property can be set to NO_LOGGING, PARTIAL_LOGGING or FULL_LOGGING. For partial logging, the full bid curve is omitted and only the minimum, maximum and effective power values are logged.
matcher.price.log.level	String	N	FULL_LOGGING	The logging level of published PowerMatcher price updates. This property can be set to NO_LOGGING or FULL_LOGGING.
agent.bid.log.level	String	N	NO_LOGGING	The logging level of the PowerMatcher bid updates published to the parent matcher. This property can be set to NO_LOGGING, PARTIAL_LOGGING or FULL_LOGGING. For partial logging, the full bid curve is omitted and only the minimum, maximum and effective power values are logged.
agent.price.log.level	String	N	NO_LOGGING	The logging level of PowerMatcher price updates received from the parent matcher. This property can be set to NO_LOGGING or FULL_LOGGING.

6.1.3 CSV Logging agent

The CSV Logging Agent receives price update and bid log messages and stores them in a comma separated file (csv-file).

Component: CSV Logging Agent				Type: factory component
Class: net.powermatcher.core.agent.logging.CSVLoggingAgent				
Dependencies:				
Log Listener Adapter (optional)				
Property	Type	Req.	Default	Description
id	String	Y	-	Component instance id e.g. 'csvlogging'
cluster.id	String	N	DefaultCluster	PowerMatcher cluster id.
enabled	boolean	N	true	If true component will not be activated when all dependencies are resolved. Otherwise if false it will never be activated.
update.interval	int	N	30	Number of seconds of message interval.
measurement.logging.pattern	String	N	'measurement_log_YYMMdd.csv'	Define the measurement logging pattern.
powermatcher.bid.logging.pattern	String	N	'pwm_bid_log_YYYYMMdd.csv'	Define the PowerMatcher bid logging pattern.
powermatcher.price.logging.pattern	String	N	'pwm_price_log_YYYYMMdd.csv'	Define the PowerMatcher bid logging pattern.
status.logging.pattern	String	N	'status_log_YYYYMMdd.csv'	Define the status logging pattern.
date.format	String	N	YYYY-MM-dd HH:mm:ss	Logging date format
list.separator	String	N	;	Separator of logged data.
bid.expiration.time	int	N	300	Number of seconds before a received update bid from an agents expires.
time.adapter.factory	String	N	timeAdapterFactory	The identifier of the adapter factory for creating the time service adapter.
scheduler.adapter.factory	String	N	schedulerAdapterFactory	The identifier of the adapter factory for creating the scheduler adapter.
log.listener.adapter.factory	String	N		The identifier of the adapter factory for creating the log listener adapter. Direct connections do not require a log listener adapter, and for this reason the default is an empty string. Fill in the default identifier "logListenerAdapterFactory" of the messaging log listener adapter factory to receive log message from the messaging bus. Multiple adapters can be configured by using a comma-separated list of factory ids.

6.1.4 Objective agent

The Objective Agent component influences the market price to achieve a static or dynamic power objective.

Component: Objective Agent	Type: factory component
Class: net.powermatcher.core.agent.objective.ObjectiveAgent	
Dependencies:	

Agent Adapter Matcher Adapter (optional) Logging Adapter (optional) Objective Adapter (optional)				
Property	Type	Req.	Default	Description
id	String	Y	-	Component instance id e.g. 'objectiveagent'
cluster.id	String	N	DefaultCluster	PowerMatcher cluster id.
enabled	boolean	N	true	If true component will not be activated when all dependencies are resolved. Otherwise if false it will never be activated.
update.interval	int	N	30	Number of seconds of message interval.
objective.bid	String	Y	(0\\,0.0):(0\\,0.0)	Defines the objective bid. Format is a sequence of pairs of price/power points (int: price, double: power).
bid.expiration.time	int	N	300	Number of seconds before a received update bid from client agents expires.
time.adapter.factory	String	N	timeAdapterFactory	The identifier of the adapter factory for creating the time service adapter.
scheduler.adapter.factory	String	N	schedulerAdapterFactory	The identifier of the adapter factory for creating the scheduler adapter.
agent.adapter.factory	String	Y	directProtocolAdapterFactory	The identifier of the adapter factory for creating the agent adapter. By default the adapter that connects directly to the matcher is used. Fill in the default identifier "agentProtocolAdapterFactory" of the messaging protocol adapter factory to publish bid updates to and receive price updates from the messaging bus. Multiple adapters can be configured by using a comma-separated list of factory ids.
matcher.id	String	Y	-	The id of the matcher to publish bids to and receive price updates from. The direct protocol adapter uses this id to locate the parent matcher to make the direct connection to. The messaging protocol adapter uses this id to address the parent matcher in the publication topic. When multiple adapters are configured for agent.adapter.factory, multiple ids must be configured here by using a comma-separated list of log listener ids. This can, for example, be used to publish bids to the auctioneer as well as to an objective agent.
matcher.adapter.factory	String	N		The identifier of the adapter factory for creating the matcher adapter. Direct connections do not require a matcher adapter, and for this reason the default is an empty string. Fill in the default identifier "matcherProtocolAdapterFactory" of the messaging protocol adapter factory to receive bid updates from and publish price updates to the messaging bus. Multiple adapters can be configured by using a comma-separated list of factory ids.
logging.adapter.factory	String	N	directLoggingAdapterFactory	The identifier of the adapter factory for creating an adapter for logging PowerMatcher bid and price events. By default the adapter that connects directly to the logging agent is used. It is also possible to configure the messaging logging adapter that publishes log messages to the bus. Multiple adapters can be configured by using a comma-separated list of factory ids.
log.listener.id	String	N	csvlogging	The id of the component that will be listening for the logging messages. The direct logging adapter uses this id to locate the component to make the direct connection to. The messaging logging adapter uses this id to address the log listener in the publication topic. When multiple adapters are configured for logging.adapter.factory, multiple ids must be configured here by using a comma-separated list of log listener ids.
matcher.agent.bid.log.level	String	N	FULL_LOGGING	The logging level of PowerMatcher bid updates received by the matcher. This property can be set to NO_LOGGING, PARTIAL_LOGGING or FULL_LOGGING. For partial logging, the full bid curve is omitted and only the minimum, maximum and effective power values are logged.
matcher.aggregated.bid.log.level	String	N	NO_LOGGING	The logging level of PowerMatcher aggregated bid of the matcher. This property can be set to NO_LOGGING, PARTIAL_LOGGING or FULL_LOGGING. For partial logging, the full bid curve is omitted and only the minimum, maximum and effective power values are logged.
matcher.price.log.level	String	N	FULL_LOGGING	The logging level of published PowerMatcher price updates. This property can be set to NO_LOGGING or FULL_LOGGING.
agent.bid.log.level	String	N	NO_LOGGING	The logging level of the PowerMatcher bid updates published to the parent matcher. This property can be set to NO_LOGGING, PARTIAL_LOGGING or FULL_LOGGING. For partial logging, the full bid curve is omitted and only the minimum, maximum and effective power values are logged.
agent.price.log.level	String	N	NO_LOGGING	The logging level of PowerMatcher price updates received from the parent matcher. This property can be set to NO_LOGGING or

				FULL_LOGGING.
objective.adapter.factory	String	N		The identifier of the adapter factory for creating an adapter for dynamically controlling the objective bid. By default no adapter is configured and the static objective.bid is used.

6.1.5 Test agent

The Test Agent is a simple example agent that is also used for testing purposes. It receives prices updates from the matcher and sends a repeating series of increasing bids based on the configured property values.

Component: Test Agent				Type: factory component
Class: net.powermatcher.core.agent.test.TestAgent				
Dependencies: Agent Adapter Logging Adapter (optional)				
Property	Type	Req.	Default	Description
id	String	Y	-	Component instance id e.g. 'testagent'
cluster.id	String	N	DefaultCluster	PowerMatcher cluster id.
enabled	boolean	N	true	If true component will not be activated when all dependencies are resolved. Otherwise if false it will never be activated.
update.interval	int	N	30	Number of seconds of message interval.
maximum.power	double	N	1000	Defines the maximum power the agent can use.
minimum.power	double	N	1000	Defines the minimum power the agent can use.
maximum.price	int	N	120	Defines the maximum bid price of the update bids the agent sends.
minimum.price	int	N	0	Defines the minimum bid price of the update bids the agent sends.
steps	int	N	12	Number of price steps (price variations) in the update bids between the minimum price and maximum price.
time.adapter.factory	String	N	timeAdapterFactory	The identifier of the adapter factory for creating the time service adapter.
scheduler.adapter.factory	String	N	schedulerAdapterFactory	The identifier of the adapter factory for creating the scheduler adapter.
agent.adapter.factory	String	Y	directProtocolAdapterFactory	The identifier of the adapter factory for creating the agent adapter. By default the adapter that connects directly to the matcher is used. Fill in the default identifier "agentProtocolAdapterFactory" of the messaging protocol adapter factory to publish bid updates to and receive price updates from the messaging bus. Multiple adapters can be configured by using a comma-separated list of factory ids.
matcher.id	String	Y	-	The id of the matcher to publish bids to and receive price updates from. The direct protocol adapter uses this id to locate the parent matcher to make the direct connection to. The messaging protocol adapter uses this id to address the parent matcher in the publication topic. When multiple adapters are configured for agent.adapter.factory, multiple ids must be configured here by using a comma-separated list of log listener ids. This can, for example, be used to publish bids to the auctioneer as well as to an objective agent.
logging.adapter.factory	String	N	directLoggingAdapterFactory	The identifier of the adapter factory for creating an adapter for logging PowerMatcher bid and price events. By default the adapter that connects directly to the logging agent is used. It is also possible to configure the messaging logging adapter that publishes log messages to the bus. Multiple adapters can be configured by using a comma-separated list of factory ids.
log.listener.id	String	N	csvlogging	The id of the component that will be listening for the logging messages. The direct logging adapter uses this id to locate the component to make the direct connection to. The messaging logging adapter uses this id to address the log listener in the publication topic. When multiple adapters are configured for logging.adapter.factory, multiple ids must be configured here by using a comma-separated list of log listener ids.
agent.bid.log.level	String	N	NO_LOGGING	The logging level of the PowerMatcher bid updates published to the parent matcher. This property can be set to NO_LOGGING, PARTIAL_LOGGING or FULL_LOGGING. For partial logging, the full bid curve is omitted and only the minimum, maximum and effective power values are logged.
agent.price.log.level	String	N	NO_LOGGING	The logging level of PowerMatcher price updates received from the parent matcher. This property can be set to NO_LOGGING or FULL_LOGGING.

6.1.6 Template agents

The Core Examples projects define two example agents that can be used as the starting point for developing new agents. Both example agents repeatedly publish a constant step-shaped bid. The second example agent extends the first agent with an example adapter that sends a message to itself.

Component: Example Agent 1			Type: factory component	
Class: net.powermatcher.core.agent.example.ExampleAgent1				
Dependencies: Agent Adapter Logging Adapter (optional)				
Property	Type	Req.	Default	Description
id	String	Y	-	Component instance id e.g. 'exampleagent1'
cluster.id	String	N	DefaultCluster	PowerMatcher cluster id.
enabled	boolean	N	true	If true component will not be activated when all dependencies are resolved. Otherwise if false it will never be activated.
update.interval	int	N	30	Number of seconds of message interval.
bid.price	double	N	0.50	Constant bid step price.
bid.power	double	N	100.0	Constant bid step demand.
time.adapter.factory	String	N	timeAdapterFactory	The identifier of the adapter factory for creating the time service adapter.
scheduler.adapter.factory	String	N	schedulerAdapterFactory	The identifier of the adapter factory for creating the scheduler adapter.
agent.adapter.factory	String	Y	directProtocolAdapterFactory	The identifier of the adapter factory for creating the agent adapter. By default the adapter that connects directly to the matcher is used. Fill in the default identifier "agentProtocolAdapterFactory" of the messaging protocol adapter factory to publish bid updates to and receive price updates from the messaging bus. Multiple adapters can be configured by using a comma-separated list of factory ids.
matcher.id	String	Y	-	The id of the matcher to publish bids to and receive price updates from. The direct protocol adapter uses this id to locate the parent matcher to make the direct connection to. The messaging protocol adapter uses this id to address the parent matcher in the publication topic. When multiple adapters are configured for agent.adapter.factory, multiple ids must be configured here by using a comma-separated list of log listener ids. This can, for example, be used to publish bids to the auctioneer as well as to an objective agent.
logging.adapter.factory	String	N	directLoggingAdapterFactory	The identifier of the adapter factory for creating an adapter for logging PowerMatcher bid and price events. By default the adapter that connects directly to the logging agent is used. It is also possible to configure the messaging logging adapter that publishes log messages to the bus. Multiple adapters can be configured by using a comma-separated list of factory ids.
log.listener.id	String	N	csvlogging	The id of the component that will be listening for the logging messages. The direct logging adapter uses this id to locate the component to make the direct connection to. The messaging logging adapter uses this id to address the log listener in the publication topic. When multiple adapters are configured for logging.adapter.factory, multiple ids must be configured here by using a comma-separated list of log listener ids.
agent.bid.log.level	String	N	NO_LOGGING	The logging level of the PowerMatcher bid updates published to the parent matcher. This property can be set to NO_LOGGING, PARTIAL_LOGGING or FULL_LOGGING. For partial logging, the full bid curve is omitted and only the minimum, maximum and effective power values are logged.
agent.price.log.level	String	N	NO_LOGGING	The logging level of PowerMatcher price updates received from the parent matcher. This property can be set to NO_LOGGING or FULL_LOGGING.

Example Agent 2 has the following properties in addition to the properties of Example Agent 1 listed above:

Component: Example Agent 2				Type: factory component	
Class: net.powermatcher.core.agent.example.ExampleAgent2					
Dependencies:					
Agent Adapter					
Logging Adapter (optional)					
Example Adapter (optional)					
Property	Type	Req.	Default	Description	

example.adapter.factory	String	N		The identifier of the adapter factory for creating an example adapter.
example.setting	String	N	default value	An example setting for the agent that is also referred to by the example adapter.

6.2 Core Component Adapters

This section describes the available adapter factories for PowerMatcher Core components.

6.2.1 Market Basis Adapter Factory

The Market Basis Adapter Factory defines the market basis for a PowerMatcher cluster. A Market Basis Adapter is created for and connected to the agent connector of the Auctioneer to publish market basis updates to the auctioneer.

Component: Market Basis Adapter Factory				Type: factory component
Class: net.powermatcher.core.agent.marketbasis.adapter.MarketBasisAdapter				
Dependencies: -				
Property	Type	Req.	Default	Description
id	String	Y	-	Factory id e.g. 'marketBasisAdapterFactory'.
cluster.id	String	N	DefaultCluster	PowerMatcher cluster id.
commodity	String	N	electricity	Commodity of this PowerMatcher cluster, for example electricity.
currency	String	N	EUR	Market currency.
market.ref	int	N	0	Initial value Market Basis sequence number (market ref) referenced by the update bids of the agents.
minimum.price	double	N	-127	Defines the lower limit of the normalized price range.
maximum.price	double	N	127	Defines the upper limit of the normalized price range.
price.steps	int	N	255	Defines the number of price steps within the normalized price range.
significance	int	N	0	Defines the significance of the agent price bids.

6.2.2 Direct Protocol Adapter Factory

The Direct Protocol Adapter Factory creates protocol adapters for a direct connection from the Agent interface of a PowerMatcher component to the matcher interface of its parent PowerMatcher component. The adapter enables the agent to send bid update messages and receive price update messages via a direct connection.

Component: Direct Protocol Adapter Factory				Type: factory component
Class: net.powermatcher.core.direct.protocol.adapter.component.DirectProtocolAdapterFactory				
Dependencies: -				
Property	Type	Req.	Default	Description
id	String	Y	-	Factory id e.g. 'directProtocolAdapterFactory'.
cluster.id	String	N	DefaultCluster	PowerMatcher cluster id.

6.2.3 Agent Protocol Adapter Factory

The Agent Protocol Adapter Factory creates messaging protocol adapters for the Agent interface of a PowerMatcher component. The adapter enables the agent to send bid update messages and receive price update messages, using the configured messaging protocol. Note that the parent matcher must have a Messaging Matcher Protocol Adapter Factory configured for the same protocol.

Component: Agent Protocol Adapter Factory				Type: factory component
Class: net.powermatcher.core.messaging.protocol.adapter.AgentProtocolAdapterFactory				
Dependencies: Messaging Connection Adapter				
Property	Type	Req.	Default	Description
id	String	Y	-	Factory id e.g. 'agentProtocolAdapterFactory'.
cluster.id	String	N	DefaultCluster	PowerMatcher cluster id.
messaging.protocol	String	Y	INTERNAL	Protocol to be used. Possible values: { "INTERNAL_v1",

			L_v1	"HAN_rev6" }
messaging.adapter.factory	String	Y	mqttv3Con nectionFac tory	The adapter factory for creating the messaging connection.
bid.topic.suffix	String	N	UpdateBid	Update bid topic name suffix. Agents will send update bid messages to this topic and the matcher will process them.
price.info.topic.suffix	String	N	UpdatePri celInfo	Price Info topic name suffix. The matcher will subscribe to this topic for price info messages from the parent matcher.

6.2.4 Matcher Protocol Adapter Factory

The Matcher Protocol Adapter Factory creates messaging protocol adapters for the Matcher interface of a PowerMatcher component. The adapter enables the matcher to receive bid updates and send price updates to child agents and matcher, using the configured messaging protocol. Note that the parent matcher must have a Messaging Agent Protocol Adapter Factory configured for the same protocol.

Component: Matcher Protocol Adapter Factory				Type: factory component
Class: net.powermatcher.core.messaging.protocol.adapter.MatcherProtocolAdapterFactory				
Dependencies: Messaging Connection Adapter				
Property	Type	Req.	Default	Description
id	String	Y	-	Factory id e.g. 'matcherProtocolAdapterFactory'.
cluster.id	String	N	DefaultCluster	PowerMatcher cluster id.
messaging.protocol	String	Y	INTERNAL_v1	Protocol to be used. Possible values: { "INTERNAL_v1", "HAN_rev6" }
messaging.adapter.factory	String	Y	mqttv3ConnectionFactory	The adapter factory for creating the messaging connection.
bid.topic.suffix	String	N	UpdateBid	Update bid topic name suffix. Agents will send update bid messages to this topic and the matcher will process them.
price.info.topic.suffix	String	N	UpdatePriceInfo	Price Info topic name suffix. The matcher will subscribe to this topic for price info messages from the parent matcher.

Note that it is possible to configure both a Direct Protocol Adapter Factory for an agent to directly connect to a matcher, as well as a Matcher Protocol Adapter Factory for the same matcher that uses a messaging connection. In that case the matcher will receive bids from some agents via a direct connection and via a messaging connection from other agents.

6.2.5 Direct Logging Adapter Factory

The Direct Logging Adapter Factory creates protocol logging adapters for a PowerMatcher component that connect directly to a logging component. The adapter enables agents and matchers to log PowerMatcher bid and price updates directly to a logging component running in the same runtime.

Component: Direct Logging Adapter Factory				Type: factory component
Class: net.powermatcher.core.direct.protocol.adapter.component.DirectLoggingAdapterFactory				
Dependencies: -				
Property	Type	Req.	Default	Description
id	String	Y	-	Factory id e.g. 'directLoggingAdapterFactory'.
cluster.id	String	N	DefaultCluster	PowerMatcher cluster id.

Note that it is possible to configure both a Direct Logging Adapter Factory as well as a Logging Adapter Factory that uses a messaging connection for the same PowerMatcher component. In that case the log messages will be published both over a direct connection as well as over a messaging connection.

6.2.6 Logging Adapter Factory

The Logging Adapter Factory creates messaging protocol logging adapters for a PowerMatcher component. The adapter enables agents and matchers to log PowerMatcher bid and price updates.

Component: Logging Adapter Factory		Type: factory component
Class: net.powermatcher.core.messaging.protocol.adapter.LoggingAdapterFactory		
Dependencies:		
Messaging Connection Adapter		

Property	Type	Req.	Default	Description
id	String	Y	-	Factory id e.g. 'loggingAdapterFactory'.
cluster.id	String	N	DefaultCluster	PowerMatcher cluster id.
connector.id	String	N		If a connector.id is specified, the messaging connection will use connector.id instead of id as the client id for the MQTT connection. This allows a single MQTT client connection to be shared, for the purpose of scalability, between multiple agents running in the same runtime environment.
messaging.adapter.factory	String	Y	mqttv3ConnectionFactory	The adapter factory for creating the messaging connection.
bid.topic.suffix	String	N	UpdateBid	Update bid topic name suffix. Agents will send update bid messages to this topic and the matcher will process them.
price.info.topic.suffix	String	N	UpdatePriceInfo	Price Info topic name suffix. The matcher will subscribe to this topic for price info messages from the parent matcher.
log.topic.suffix	String	N	Log	Log topic name suffix. Agent will use this suffix to extend the bid and price topics for the logging topic.

Note that it is possible to configure both a Direct Logging Adapter Factory as well as a Logging Adapter Factory that uses a messaging connection for the same PowerMatcher component. In that case the log messages will be published both over a direct connection as well as over a messaging connection.

6.2.7 Log Listener Adapter Factory

The Logging Adapter Factory creates messaging protocol logging adapters for a logging component. The adapter enables a logging component to receive PowerMatcher bid and price log messages.

Component: Log Listener Adapter Factory				Type: factory component
Class: net.powermatcher.core.messaging.protocol.adapter.LogListenerAdapterFactory				
Dependencies:				
Messaging Connection Adapter				
Property	Type	Req.	Default	Description
id	String	Y	-	Factory id e.g. 'logListenerAdapterFactory'.
cluster.id	String	N	DefaultCluster	PowerMatcher cluster id.
messaging.adapter.factory	String	Y	mqttv3ConnectionFactory	The adapter factory for creating the messaging connection.
bid.topic.suffix	String	N	UpdateBid	Update bid topic name suffix. Agents will send update bid messages to this topic and the matcher will process them.
price.info.topic.suffix	String	N	UpdatePriceInfo	Price Info topic name suffix. The matcher will subscribe to this topic for price info messages from the parent matcher.
log.topic.suffix	String	N	Log	Log topic name suffix. Agent will use this suffix to extend the bid and price topics for the logging topic.

Note that it is possible to configure both a Direct Logging Adapter Factory for an agent or matcher to directly connect to a logging component, as well as a Log Listener Adapter Factory for the same logging component that uses a messaging connection. In that case the logging component will logging events from some agents via a direct connection and via a messaging connection from other agents.

6.2.8 MQTT v3 Connection Adapter Factory

The MQTT v3 Connection Adapter Factory creates messaging connections for adapters that use messaging. The adapter implements the MQTT version 3 protocol. The MQTTv3 client connection that is created by the adapter factory is shared between adapters that have the same connector.id (the default for the connector.id is the id of the component).

Component: MQTT v3 Connection Adapter Factory			Type: factory component	
Class: net.powermatcher.core.messaging.mqttv3.Mqttv3ConnectionFactory				
Dependencies: -				
Property	Type	Req.	Default	Description
id	String	Y	-	Factory id e.g. 'mqttv3ConnectionFactory'.
cluster.id	String	N	DefaultCluster	PowerMatcher cluster id.
broker.uri	String	N	tcp://localhost:1883	Broker URI.
broker.username	String	N	-	Broker username. Only applicable when broker security is enabled.
broker.password	String	N	10	Broker password. Only applicable when

notification.enabled	boolean	N	false	broker security is enabled.
hostname	String	N	-	Send notification messages.
component.name	String	N	-	Hostname of the MQTT client.
notification.topic	String	N	Status/Agent	Component name.
notification.connected.message	String	N	CONNECTED\\,\\,\\,{cluster.id}\\,\\,{id}\\,\\,{component.name}\\,\\,{component.name}\\,\\,{host.name}1883	Topic name notification messages.
notification.cleandisconnectmessage	String	N	CLEANDISCONNECTED\\,\\,\\,{cluster.id}\\,\\,{id}\\,\\,{component.name}\\,\\,{component.name}\\,\\,{host.name}	Format of notification message.
notification.disconnected.message	String	N	DISCONNECTED\\,\\,\\,{cluster.id}\\,\\,{id}\\,\\,{component.name}\\,\\,{component.name}\\,\\,{host.name}	Format of clean disconnect message.
reconnect.interval	int	N	10	Format of disconnect message caused by an exception.
				Retry interval to (re)connect.

6.2.9 Time Adapter Factory

The Time Adapter Factory creates adapters for components that require a time source.

The Time Adapter Factory creates a time source that returns the current time and runs in real time.

Component: Time Adapter Factory				Type: factory component
Class: net.powermatcher.core.scheduler.TimeAdapterFactory				
Dependencies: -				
Property	Type	Req.	Default	Description
id	String	Y	-	Factory id e.g. 'timeAdapterFactory'.
cluster.id	String	N	DefaultCluster	PowerMatcher cluster id.

6.2.10 Scheduler Adapter Factory

The Time Adapter Factory creates adapters for components that require a schedule to execute scheduled activities. The Scheduler Adapter Factory creates a scheduler that executes activities on a thread pool that has one thread for each CPU core available to the runtime. Each PowerMatcher cluster, even when deployed on the same runtime, has its own thread pool.

Component: MQTT v3 Connection Adapter Factory				Type: factory component
Class: net.powermatcher.core.scheduler.SchedulerAdapterFactory				
Dependencies: -				
Property	Type	Req.	Default	Description
id	String	Y	-	Factory id e.g. 'schedulerAdapterFactory'.
cluster.id	String	N	DefaultCluster	PowerMatcher cluster id.

6.3 Configuration Manager component

The Configuration Manager component reads OSGi configuration specifications in XML format and invokes the ConfigAdmin service of OSGi to create, update or delete configuration objects. Configuration files can be located locally, or on a remote server accessed via http or https. Basic authentication using username and password is supported for remote access.

The component checks with a configurable interval if the configuration has been updated.

Component: ConfigManager			Type: factory component	
Class: net.powermatcher.core.config.management.agent.ConfigManager				
Dependencies: -				
Property	Type	Req.	Default	Description
id	String	Y	-	Component instance id e.g. 'configmgr'
cluster.id	String	N	DefaultCluster	PowerMatcher cluster id.
node.id	String	Y	defaultnode	Id identifying the OSGi runtime node.
configuration.data.url	String	Y	file:conf-local/default_config.xml	URL identifying the location of the XML configuration resource.
configuration.data.userid	String	N	-	User id for retrieving a secured XML configuration resource.
configuration.data.password	String	N	-	Password for retrieving a secured XML configuration resource.

update.interval	int	Y	300	Refresh interval seconds for refreshing the configuration.
-----------------	-----	---	-----	--

7. Appendix B: OSGi runtime installation and setup

In order to run the PowerMatcher applications you need to set up an OSGi runtime environment. Pax-Runner can be used to simplify this task. Simply stated, Pax Runner is an OSGi framework launcher that supports the setup of different OSGi framework implementations like Apache Felix, Equinox and Knopflerfish. In this instruction we will use the Equinox implementation but will enhance the functionality by adding components from the Apache Felix implementation.

7.1.1 Pax Runner installation and setup

Follow the next steps to create a basic Pax Runner environment that will use the Equinox OSGi implementation:

1. First download the Pax Runner from the OPS4J site [2]. You may download the assembly which already contains example run scripts for Windows and Unix or you can just download just the pax-runner-x.y.z.jar file (where x.y.z is the version number). The Pax-Runner version used in the project is version 1.7.6.
2. Create a **target directory** which to store the Pax Runner distribution and the PowerMatcher bundles. In this document we assume the directory is C:\apache\pax-runner.
3. In the Pax Runner target directory create the following directories:

Directory name	Purpose
required-bundles	Required non-PowerMatcher bundles. For example for logging, web console, administration and configuration
bundles	This will contain the application specific bundles.
conf	Directory for configuration files

4. Copy the Pax Runner jar file to the target directory.
5. Create a property file named 'platform.properties' in the Pax Runner target directory. This file will contain the start-up environment variables or system properties. In one of the next steps you will edit this file.
6. In the Pax Runner directory Create a shell script named 'run.sh' for starting PaxRunner. For windows create a file named 'run.bat' that will contain the following:

```
call java -jar pax-runner-1.7.6.jar --platform=equinox --version=3.7.1 --ups --
workingDirectory=. scan-dir:required-bundles scan-file:file:platform.properties
```

This will start Pax Runner using the Equinox OSGi implementation version 3.7.1 from the current directory as a working directory. The scan:dir and scan:file parameters direct the Pax Runner to load bundles or settings from a specific directory or file.

During start-up Pax-Runner will download the specified OSGi framework (.jar file), which requires a connection to the Internet. If this is not available, you first can download it from the Eclipse site using another machine that does have Internet access [3].

The Pax Runner setup is ready. You can start it using the run command script. It will start an OSGi environment with a web console. In the platform properties we specified the port number for the web console, this will make it accessible from (after installing the required bundle):

<http://localhost:8080/system/console>

We have created now a base OSGi environment and we have to prepare it for running PowerMatcher based applications. Follow the steps described in the next sections.

7.1.2 Install the prerequisite bundles

A PowerMatcher application in OSGi requires a set of libraries. The `required-bundles` directory will contain all the bundle jars that are required. Some prerequisite libraries are not delivered with the release, but have to be acquired individually. Copy the jar libraries listed in Table 5 to this directory:

Function	JAR file	Description	Download
OSGi	osgi.cmpn-4.2.0.jar	OSGi Service Platform Service Compendium bundle. Normally it should not be necessary and it is undesirable to install the Compendium bundle, as it is best practice for bundles implementing Compendium services to export the API package. However, not all service implementations follow this practice.	See [4].
Eclipse Equinox OSGi implementation	org.eclipse.equinox.common.jar org.eclipse.equinox.preferences.jar org.eclipse.equinox.util.jar	Individual bundles that are prerequisites for MicroBroker in PowerMatcher Extension. For PowerMatcher Core this step can be skipped.	See [3], choose build version then Add-on bundles and find specified jar.
Base 64	org.apache.commons.codec	Encoders and decoders such as Base64	See [5].
Apache Felix web console	org.apache.felix.webconsole-3.1.8.jar org.apache.felix.http.jetty-2.2.0.jar org.apache.felix.scr-1.6.0.jar org.apache.felix.metatype-1.0.4.jar	Apache Felix Web Console Apache Felix Http Service Jetty Apache Felix SCR (Declarative Services) Apache Felix Metatype	[6]. Select 'Downloads' link.
Bnd libraries	biz.aQute.bndlib biz.aQute.junit	Bnd base library. Bundle that provides junit and supports testing within the OSGi framework.	See [7].
Logging	slf4j.api-1.7.2.jar ch.qos.logback.classic-1.0.9.jar ch.qos.logback.core-1.0.9.jar	Simple Logging Facade for Java (SLF4J) Logging framework. Logging framework.	See [8]. See [9].
MQTT client	org.eclipse.paho.client.mqttv3	Open source implementation of messaging client using the MQTT version 3 protocol.	See [10]

Table 5 PowerMatcher required libraries for OSGi.

Note that when you set up a development environment when the configuration project 'cnf' is created by Bndtools the libraries listed in the table have either been downloaded by Bndtools in the repository, or have been added when setting up the workspace. You can copy the libraries from the workspace when you set up your runtime.

7.1.3 Install the PowerMatcher bundles

Next, copy all PowerMatcher Core bundle jars into the `required-bundles` directory.

JAR file	Description
net.powermatcher.core.adapter.jar	PowerMatcher adapter implementation
net.powermatcher.core.adapter.component.jar	PowerMatcher adapter bundle jar
net.powermatcher.core.adapter.service.jar	PowerMatcher adapter api
net.powermatcher.core.agent.auctioneer.component.jar	Auctioneer bundle jar.
net.powermatcher.core.agent.auctioneer.jar	Auctioneer library.
net.powermatcher.core.agent.concentrator.component.jar	Concentrator bundle jar.
net.powermatcher.core.agent.concentrator.jar	Concentrator library.
net.powermatcher.core.agent.jar	PowerMatcher framework library.
net.powermatcher.core.agent.logging.component.jar	Logging bundle jar.
net.powermatcher.core.agent.logging.jar	PWM Logging library.
net.powermatcher.core.agent.marketbasis.adapter.component.jar	Market basis adapter bundle.
net.powermatcher.core.agent.marketbasis.adapter.jar	Market basis adapter implementation.
net.powermatcher.core.agent.objective.component.jar	Objective agent bundle jar.
net.powermatcher.core.agent.objective.jar	Objective agent implementation.
net.powermatcher.core.agent.test.component.jar	Test agent bundle jar.
net.powermatcher.core.agent.test.jar	Test agent implantation.
net.powermatcher.core.config.management.agent.jar	Config Manager implementation and bundle jar.
net.powermatcher.core.config.management.agent.test.jar	ConfigManager test packages.
net.powermatcher.core.configurable.jar	PWM framework configurable object.

net.powermatcher.core.direct.protocol.adapter.jar	Direct protocol adapter bundle jar.
net.powermatcher.core.direct.protocol.adapter.component.jar	Direct protocol implementation.
net.powermatcher.core.logging.bridge.jar	Log manager.
net.powermatcher.core.messaging.framework.jar	PWM Messaging framework.
net.powermatcher.core.messaging.framework.test.jar	PWM Messaging framework test classes.
net.powermatcher.core.messaging.mqttv3.component.jar	MQTT v3 bundle jar.
net.powermatcher.core.messaging.mqttv3.jar	MQTT v3 wrapper implementation.
net.powermatcher.core.messaging.protocol.adapter.component.jar	Messaging protocol adapter bundle jar.
net.powermatcher.core.messaging.protocol.adapter.jar	Messaging protocol implementation.
net.powermatcher.core.scheduler.jar	Time and scheduler implementation.
net.powermatcher.core.scheduler.component.jar	Time and scheduler bundle jar.
net.powermatcher.core.scheduler.service.jar	Time and scheduler api.
net.powermatcher.core.xml.standard.parser.jar	PWM XML parser library

Table 6 Application bundles for PowerMatcher Core applications.

When running the PowerMatcher Core Example agent and adapter, the Core Example bundle jars should also be added to the `required-bundles` directory.

7.1.4 Configure the run time properties

Edit the property file named 'platform.properties' in the Pax Runner target directory. Add the following start-up environment variables or system properties:

```
-Dorg.apache.felix.http.nio=true
-Dorg.apache.felix.http.enable=true
-Dorg.osgi.service.http.port=8080
-Dorg.apache.felix.https.enable=true
-Dorg.osgi.service.http.port.secure=8444
-Dorg.apache.felix.http.debug=false
-Dfelix.webconsole.username=admin
-Dfelix.webconsole.password=admin
-Dlogback.configurationFile=logback.xml
-Dnet.powermatcher.core.config.management.agent.id=pwmcorecm
-Dnet.powermatcher.core.config.management.agent.node.id=pwm-core-minimal
-Dnet.powermatcher.core.config.management.agent.configuration.data.url=
file:conf/pwm_core_minimal.xml
-Dnet.powermatcher.core.config.management.agent.cluster.id=demo
-Dnet.powermatcher.core.config.management.agent.update.interval=300
```

Each line in the file will assign an environment variable in the shell session where Pax-Runner is started. The value will be assigned to a configuration of the type that is specified in the variable name. The syntax is:

```
-D<component class name>.<property name>
```

The web console username and password are the credentials that you have to specify when you open the Apache Felix web console and the port is 8080 (<http://localhost:8080/system/console>).

7.1.5 Logback logging configuration

Logging is configured by a Logback configuration file. The name of the file is specified in the `-Dlogback.configurationFile` property in the platform.properties file. The default provided configuration file creates a console appender and a daily rolling file appender which only log at error level. Refer to [9] for customizing the logging preferences.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--PowerMatcher Core - Logging configuration -->
<configuration>
  <!-- Console appender -->
  <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%d{HH:mm:ss,SSS} %c{1} - %m%n</pattern>
    </encoder>
    <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
      <level>INFO</level>
    </filter>
  </appender>

  <!--Daily rolling file appender -->
  <appender name="LOGFILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <File>demoapp.log</File>
    <Append>true</Append>
    <encoder>
      <pattern>%d{dd MMM yyyy HH:mm:ss,SSS} [%t] %-5p %c - %m%n</pattern>
    </encoder>
```

```

    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
      <FileNamePattern>demoapp.%d{yyyy-MM-dd}.log</FileNamePattern>
    </rollingPolicy>
    <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
      <level>INFO</level>
    </filter>
  </appender>

  <root level="ERROR">
    <appender-ref ref="CONSOLE" />
    <appender-ref ref="LOGFILE" />
  </root>
</configuration>

```

7.1.6 Install and run the MQTT broker

Installing and running an MQTT broker is optional and is only required if messaging-type adapters are configured. See paragraph 5.5 *MQTT broker installation*.

7.1.7 Configure the PowerMatcher application

The next step is configuring your application. Read section 4.1 for details. If you wish to create a configuration manually using the Web console, skip to the next section.

Otherwise, create an XML configuration as described in section 4.1.2 and copy the configuration file to the 'conf' directory in the target directory. Make sure that the following properties in the platform.properties file match those of your XML configuration file:

```

-Dnet.powermatcher.core.config.management.agent.node.id=sample-node
-Dnet.powermatcher.core.config.management.agent.configuration.data.url=
file:conf/pwm_core_minimal.xml
-Dnet.powermatcher.core.config.management.agent.cluster.id=DemoPwmCore

```

7.1.8 Starting the OSGi environment

Start your OSGi application with the run script of section 7.1.1.

8. Appendix C: Configuration XML schema

```
<?xml version="1.0" encoding="UTF-8"?><xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="nodeconfig">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="configuration" maxOccurs="unbounded"
          minOccurs="0" />
      </xsd:sequence>
      <xsd:attribute name="id">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:minLength value="1"></xsd:minLength>
            <xsd:maxLength value="254"></xsd:maxLength>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
      <xsd:attribute name="name" use="optional">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:maxLength value="254"></xsd:maxLength>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
      <xsd:attribute name="description" use="optional">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:maxLength value="254"></xsd:maxLength>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
      <xsd:attribute name="date" type="xsd:string" use="optional"/>
      <xsd:attribute name="updated" use="optional">
        <xsd:simpleType>
          <xsd:restriction base="xsd:int">
            <xsd:minInclusive value="0"></xsd:minInclusive>
            <xsd:maxInclusive value="1"></xsd:maxInclusive>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="property">
    <xsd:complexType>
      <xsd:attribute name="value" use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:maxLength value="3072"></xsd:maxLength>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
      <xsd:attribute name="type" use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:enumeration value="String"></xsd:enumeration>
            <xsd:enumeration value="Boolean"></xsd:enumeration>
            <xsd:enumeration value="Integer"></xsd:enumeration>
            <xsd:enumeration value="Float"></xsd:enumeration>
            <xsd:enumeration value="Double"></xsd:enumeration>
            <xsd:enumeration value="Long"></xsd:enumeration>
            <xsd:enumeration value="Short"></xsd:enumeration>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
      <xsd:attribute name="name" type="xsd:string"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="configuration">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="0"
          ref="property" />
        <xsd:element maxOccurs="unbounded" minOccurs="0"
          ref="configuration" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

```

</xsd:sequence>
<xsd:attribute name="cluster" use="required">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:maxLength value="254"/></xsd:maxLength>

    </xsd:restriction>
  </xsd:simpleType>
</xsd:attribute>
<xsd:attribute name="id" use="required">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:minLength value="1"/></xsd:minLength>
      <xsd:maxLength value="254"/></xsd:maxLength>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:attribute>
<xsd:attribute name="type" use="required">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="factory"/></xsd:enumeration>
      <xsd:enumeration value="singleton"/></xsd:enumeration>
      <xsd:enumeration value="group"/></xsd:enumeration>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:attribute>
<xsd:attribute name="pid">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:minLength value="1"/></xsd:minLength>
      <xsd:maxLength value="254"/></xsd:maxLength>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:attribute>

  <xsd:attribute name="template" type="xsd:boolean" use="optional"/></xsd:attribute>
</xsd:complexType>
</xsd:element>
</xsd:schema>

```

9. REFERENCES

[1]	Eclipse downloads, IDE for Java Developers, http://www.eclipse.org/downloads/
[2]	OPS4J site for Pax-Runner documentation and downloads: http://team.ops4j.org/wiki/display/paxrunner/Pax+Runner
[3]	Eclipse – Equinox Downloads, http://download.eclipse.org/equinox/
[4]	OSGi Compendium library download, http://mvnrepository.com/artifact/org.osgi/org.osgi.compendium
[5]	Apache Commons Code, http://commons.apache.org/codecs/
[6]	Apache Felix, http://felix.apache.org
[7]	Bndtools. For reference and documentation of Bnd: http://www.aqute.biz/Bnd/Bnd For downloads of Bndtools with Bnd, see http://bndtools.org
[8]	SLF4J - Simple Logging Facade for Java , http://www.slf4j.org/
[9]	Logback – Java logging framework, http://logback.qos.ch/
[10]	Eclipse Paho, open source implementation of open and standard messaging protocols, http://www.eclipse.org/paho/
[11]	Mosquitto, An Open Source MQTT v3.1 Broker, http://mosquitto.org/