APSConfigService

User Guide

Version: 0.9.0

Author: Tommy Svensson

Table of Contents

1 APSConfigService 1
1.1 Configuration Environments 1
1.2 Making a config class 1
1.2.1 The config values 2
1.2.2 The config annotations 2
1.2.2.1 @APSConfigDescription 2
1.2.2.2 @APSConfigItemDescription 2
1.2.2.3 @APSDefaultValue 3
1.2.3 Auto managed configurations 3
1.2.3.1 Variant 1: A simple non instantiated static member of config model type 4
1.2.3.2 Variant 2: A static instantiated ManagedConfig<ConfigModel> member. 4
1.3 API Usages 5
1.3.1 The configuration service usage 5
1.3.2 The configuration admin service usage 5
1.4 A word of advice 5
1.5 APSConfigAdminWeb screenshots 5

1 APSConfigService

This is not the simple standard OSGi service configurations, but more an application config that can also be used for services. It supports structured configurations including lists of items and lists of subconfigurations. Code that uses the configuration provide one or more configuration classes with config items. These are registered with the config service, which makes them editable/publishable though and admin web app. After registration an instance of the config can be gotten containing published or defaul values. Alternatively the config class is specified with a fully qualified name in the *APS-Configs:* MANIFEST.MF entry. In this case the configuration service acts as an extender and automatically registers and provides an instance of the config for you, without having to call the config service.

1.1 Configuration Environments

The APSConfigService supports different configuration environments. The idea is to define one config environment per installation. Configuration values can either be configuration environment specific or the same for all environments. See @ConfigItemDescription below for more information on specifying configuration environment specific values.

1.2 Making a config class

Here is an example:

```
@APSConfigDescription(
        version="1.0",
        configId=”se.natusoft.aps.exmple.myconfig”,
        group=”examples”,
        description=”An example configuration model”
)
public class MyConfig extends APSConfig {

    @APSConfigItemDescription(
        description=”Example of simple value.”,
    )
    public APSConfigValue simpleValue;

    @APSConfigItemDescription(
        description=”Example of list value.”
    )
    public APSConfigValueList listValue;

    @APSConfigItemDescription(
        description=”One instance of MySubConfig model.”
    )
    public MySubConfig mySubConfig;

    @APSConfigItemDescription(
        description=”Multiple instances of MySubConfig model.”
    )
    public APSConfigList<MySubConfig> listOfMySubConfigs;

    @APSConfigDescription(
        version=”1.0”,
        configId=”se.natusoft.aps.example.myconfig.mysubconfig”,
        description=”Example of a subconfig model. Does not have to be inner class!”
    )
    public static class MySubConfig extends APSConfig {

        @APSConfigItemDescription(
            description=”Description of values.”
        )
        public APSConfigValueList listOfValues;

        @APSConfigItemDescription(
            description=”Description of another value.”
        )
```

```
        public APSConfigValue anotherValue;
    }
}
```

1.2.1 The config values

Now you might be wondering, why not an interface, and why *public* and why *APSConfigValue*, *APSConfigValueList*, and *APSConfigList*?

The reason for not using an interface and provide a java.lang.reflect.Proxy implementation of it is that OSGi has separate class loaders for each bundle. This means a service cannot proxy an interface provided by another bundle. Well, there are ways to go around that, but I did not want to do that unless that was the only option available. In this case it wasn't. Therefore I use the above listed APS*Value classes as value containers. They are public so that they can be accessed and set by the APSConfigService. When you get the main config class instance back from the service all values will have valid instances. Each APS*Value has an internal reference to its config value in the internal config store. So if the value is updated this will be immediately reflected since it is referencing the one and only instance of it in the config store.

All config values are strings! All config values are stored as strings. The **APSConfigValue** container however have *toBoolean()*, *toDate()*, *toDouble()*, *toFloat()*, *toInt()*, *toLong()*, *toByte()*, *toShort()*, and *toString()* methods on it.

The **APSConfigList<Type>** container is an *java.lang.Iterable* of <Type> type objects. The <Type> cannot however be anything. When used directly in a config model it must be <Type extends APSConfig>. That is, you can only specify other config models extending APSConfig. The only exception to that is **APSConfigValueList** which is defined as:

```
public interface APSConfigValueList extends APSConfigList<APSConfigValue> {}
```

- Use **APSConfigValue** for plain values.
- Use **APSConfigValueList** for a list of plain values.
- Use **MyConfigModel extends APSConfig** for a subconfig model.
- Use **APSConfigList<MyConfigModel extends APSConfig>** for a list of subconfig models.

1.2.2 The config annotations

The following 3 annotations are available for use on configuration models.

1.2.2.1 @APSConfigDescription

```
@APSConfigDescription(
    version="1.0",
    configId=”se.natusoft.aps.exmple.myconfig”,
    group=”docs.examples”,
    description=”An example configuration model”
)
```

This is an annotation for a configuration model.

version - The version of the config model. This is required.

configId - The unique id of the configuration model. Use same approch as for packages. This is required.

group - This specifies a group or rather a tree branch that the config belongs under. This is only used by the configuration admin web app to render a tree of configuration models. This is optional.

description - This describes the configuration model.

1.2.2.2 @APSConfigItemDescription

```
@APSConfigItemDescription(
    description=”Example of simple value.”,
    datePattern=”yyMMdd”,
    environmentSpecific=true/false,
    isBoolean=true/false,
    validValues={”high”, ”medium”, ”low”},
)
```

This is an annotation for a configuration item whithin a configuration model.

description - This describes the configuration value. The configuration admin web app uses this to explain the configuration value to the person editing the configuration. This is required.

datePattern - This is a date pattern that will be passed to SimpleDateFormat to convert the date in the string value to a java.util.Date object and is used by the *toDate()* method of APSConfigValue. This date format will also be displayed in the configuration admin web app to hint at the date format to the person editing the configuration. The configuration admin web app will also use a calendar field if this is available. The calendar field has a complete calendar popup that lets you choose a date. This is optional.

environmentSpecific - This indicates that the config value can have different values depending on which config environment is active. This defaults to false in which case the value will apply to all config environments. This is optional.

isBoolean - This indicates that the config value is of boolean type. This is used by the configuration admin web app to turn this into a checkbox rather than a text field. This defaults to false and is this optional.

validValues - This is an array of strings ({”...”, ..., ”...”}) containing the only valid values for this config value. This is used by the configuration admin web app to provide a dropdown menu of the alternatives rather than a text field. This defaults to {} and is thus optional.

defaultValue - This is an array of @APSDefaultValue annotations. Se the description of this annotation below. This allows not only for providing a default value, but for providing a default value per config environment (which is why there is an array of @APSDefaultValue annotations!). Thus you can deliver pre configured configuration for all configuration environments. If a config environment is not specified for a default value then it applies for all configuration environments. Some configuration values are better off without default values, like hosts and ports for other remote services. The application/server maintenance people responsible for an installation in general knows this information better than the developers.

1.2.2.3 @APSDefaultValue

```
@APSDefaultValue {
    configEnv=”production”,
    value=”15”
}
```

configEnv - This specifies the configuration environment this default value applies to. ”default” means all/any configuration environment and is the default value if not specified.

value - This is the default value of the configuration value for the configuration environment specified by configEnv.

1.2.3 Auto managed configurations

It is possible to let the APSConfigService act as an extender and automatically register and setup config instances on bundle deploy by adding the **APS-Configs:** MANIFEST.MF header and a comma separated list of fully qualified names of config models. There are two variants of how to define the auto managed instance.

Warning: Auto managed configurations cannot ever be accessed during bundle activation in default activation thread! If the activation code starts a new thread then it is OK to access auto managed configuration in that thread, but only with variant 2! (the thread have to put itself to sleep until the configuration becomes managed. This is described below).

1.2.3.1 Variant 1: A simple non instantiated static member of config model type

Example:

```
    @APSConfigDescription(
        version="1.0",
        configId=”se.natusoft.aps.exmple.myconfig”,
        group=”examples”,
        description=”An example configuration model”
    )
    public class MyConfig extends APSConfig {

    –>  public static MyConfig myConfig;  <–

        @APSConfigItemDescription(
            description=”Example of simple value.”,
        )
        public APSConfigValue simpleValue;

        @APSConfigItemDescription(
            description=”Example of list value.”
        )
        public APSConfigValueList listValue;
        ...
```

To access this variant of managed config do:

```
    MyConfig.myConfig.simpleValue.toString()/toInt()/toDouble()/...
```

A warning: This variant does not provide any support for determining if the configuration has become managaged yet. If you access it to early it will be null. Therefore you should only use this variant if you know it will become managed before it is referenced. The other variant allows you to check and wait for a config to become managed.

1.2.3.2 Variant 2: A static instantiated ManagedConfig<ConfigModel> member.

Example:

```
    @APSConfigDescription(
        version="1.0",
        configId=”se.natusoft.aps.exmple.myconfig”,
        group=”examples”,
        description=”An example configuration model”
    )
    public class MyConfig extends APSConfig {

    –>  public static final ManagedConfig<MyConfig> managed = new
ManagedConfig<MyConfig>();  <–

        @APSConfigItemDescription(
            description=”Example of simple value.”,
        )
        public APSConfigValue simpleValue;

        @APSConfigItemDescription(
            description=”Example of list value.”
        )
        public APSConfigValueList listValue;
        ...
```

There is a possibility that code started in a bundle, especially threads might start running before the config has become managed. In such cases the following will solve that:

```
    if (!MyConfig.managed.isManaged()) {
        MyConfig.managed.waitUntilManaged();
    }
```

Do not ever do this during start() of a Bundle activator! That would cause a never ending dead-lock!

To access this variant of managed config do:

```
    MyConfig.managed.get().simpleValue.toString()/toInt()/toDouble()/...
```

1.3 API Usages

1.3.1 The configuration service usage

The APSConfigService API looks like this:

```
    public interface APSConfigService {
        void registerConfiguration(Class<? extends APSConfig> configClass, boolean
forService) throws APSConfigException;
        void unregisterConfiguration(Class<? extends APSConfig> configClass);
        <Config extends APSConfig> Config getConfiguration(Class<Config> configClass)
throws APSConfigException;
    }
```

On bundle start you register the configuration. On bundle stop you unregister it. Inbetween you access it. It is a good idea to call getConfiguration(...) after register on bundle start and the pass this instance to your services, etc.

If the *forServices* flag is *true* then this configuration will also be registered in the standard OSGi configuration service. Please be warned however that APSConfigService stores its configuration values in properties files, but with rather complex keys. For non structured, flat configurations it might make some sense to register it with the standard osgi service also, but in most cases there is no point in doing this. I’m not even sure why I have this option!

Please note that if you are using managed configs (see above) then you never need to call this service API, not even lookup/track the APSConfigService!

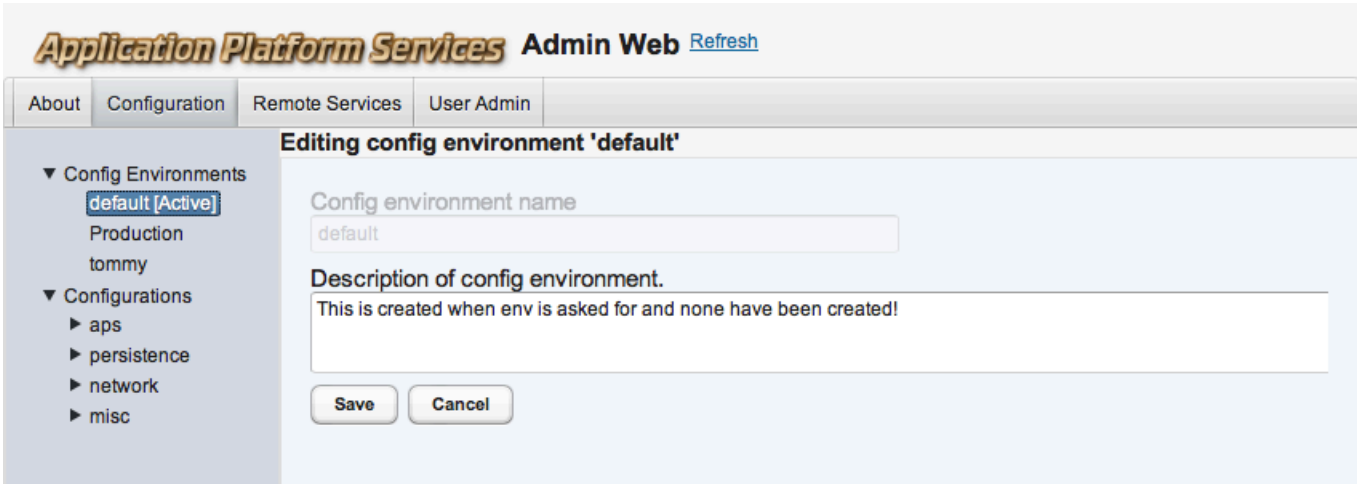
1.3.2 The configuration admin service usage

The APSconfigAdminService only needs to be used if you implement a configuration editor. APSConfigAdminWeb uses this API for example. See the javadoc for the API.

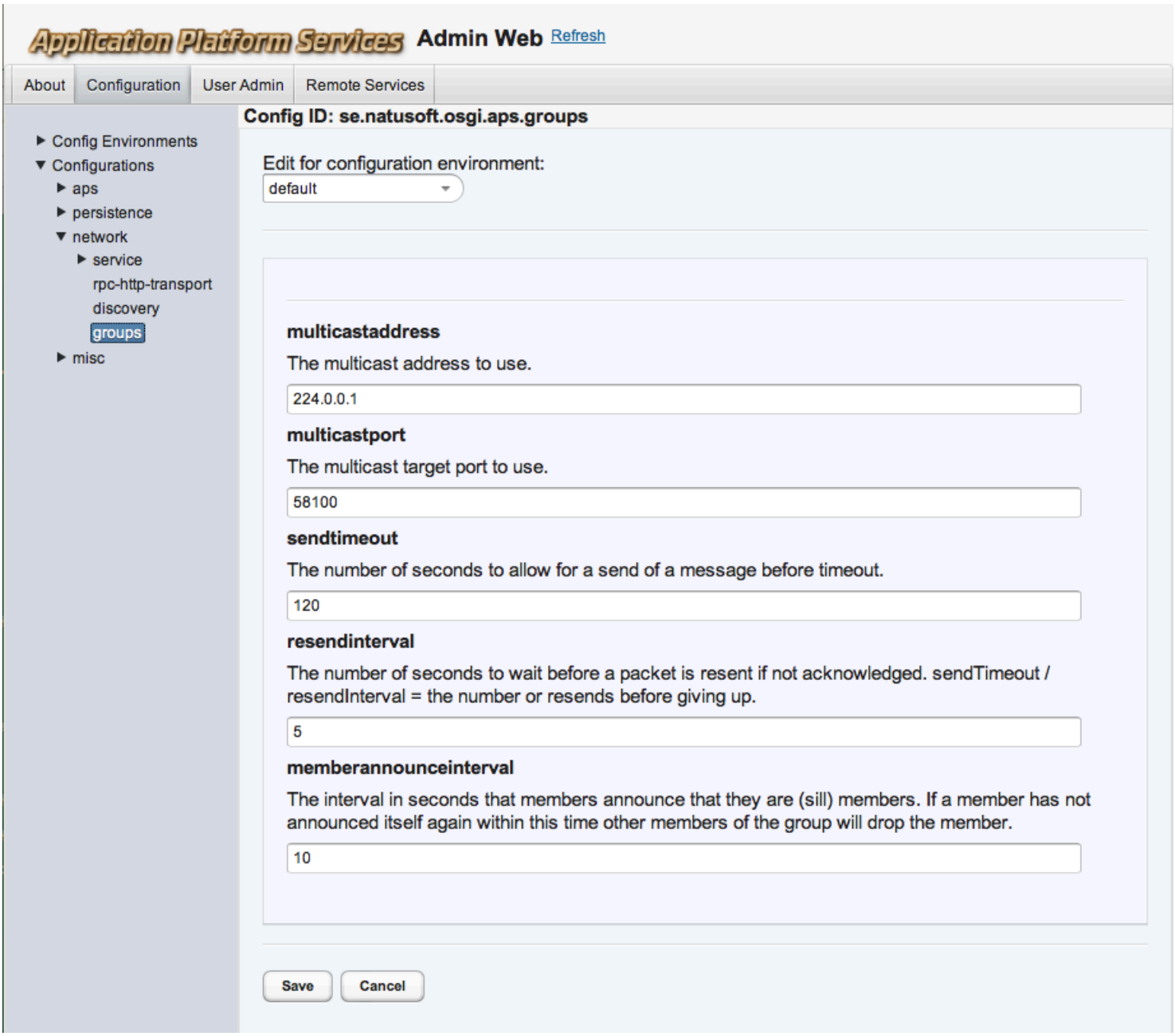
1.4 A word of advice

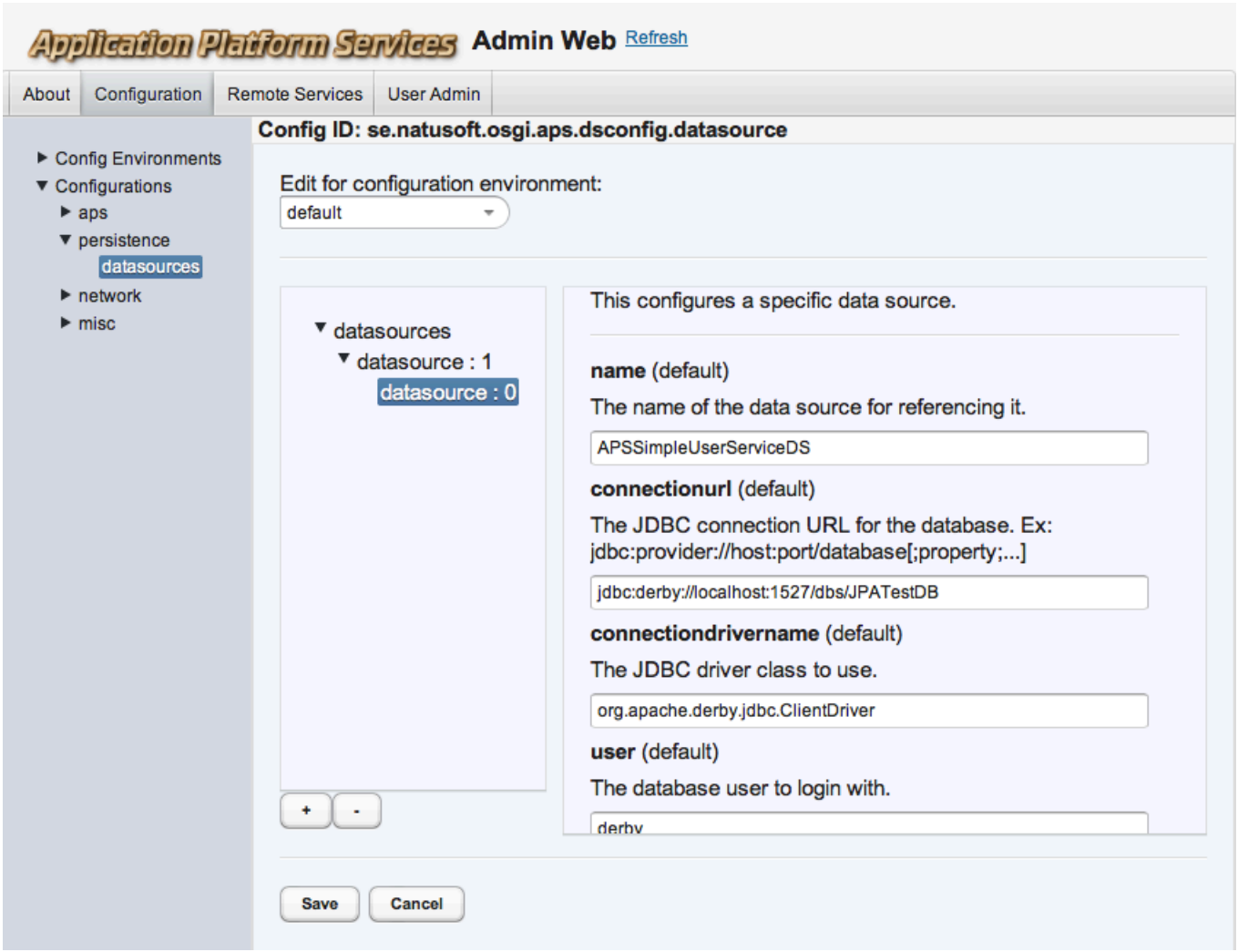
It is quite possible to make config structures of great complexity. **DON'T!** Even if it seems manageable from a code perspective it might not be that from a admin perspective. Keep it simple always apply!

1.5 APSConfigAdminWeb screenshots


Application Platform Services
Admin Web
Refresh
About
Configuration
Remote Services
User Admin
Editing config environment 'default'
Config Environments
default [Active]
Production
tommy
Configurations
aps
persistence
network
misc
Config environment name
default
Description of config environment.
This is created when env is asked for and none have been created!
Save
Cancel


Application Platform Services
Admin Web
Refresh
About
Configuration
Remote Services
User Admin
Config Environments
default [Active]
Production
tommy
Configurations
Configuration Environments
There are usually several deployment environments where some configurations do differ per environment. The APS configurations supports different values for different environments. This is the place where you define environments. A common scenario is development, systemtest, acceptancetest, and production. In some cases there might be several of each. Example: syst1, syst2, syst3.
To create a new configuration environment right-click on "Config Environments" and select "New config env".
New config env
To select one configuration environment as active right-click on it in the menu and select "Set as active".
Set as active
Delete config env
To delete a configuration environment right-click on it in the menu and select "Delete config env".
Set as active
Delete config env


Application Platform Services
Admin Web
Refresh
About
Configuration
User Admin
Remote Services
Config ID: se.natusoft.osgi.aps.groups
Config Environments
Configurations
aps
persistence
network
service
rpc-http-transport
discovery
groups
misc
Edit for configuration environment:
default
multicastaddress
The multicast address to use.
224.0.0.1
multicastport
The multicast target port to use.
58100
sendtimeout
The number of seconds to allow for a send of a message before timeout.
120
resendinterval
The number of seconds to wait before a packet is resent if not acknowledged. sendTimeout / resendInterval = the number or resends before giving up.
5
memberannounceinterval
The interval in seconds that members announce that they are (sill) members. If a member has not announced itself again within this time other members of the group will drop the member.
10
Save
Cancel


Application Platform Services
Admin Web
Refresh
About
Configuration
Remote Services
User Admin
Config ID: se.natusoft.osgi.aps.dsconfig.datasource
Config Environments
Configurations
aps
persistence
datasources
network
misc
Edit for configuration environment:
default
datasources
datasource : 1
datasource : 0
+
-
This configures a specific data source.
name (default)
The name of the data source for referencing it.
APSSimpleUserServiceDS
connectionurl (default)
The JDBC connection URL for the database. Ex: jdbc:provider://host:port/database[;property;...]
jdbc:derby://localhost:1527/dbs/JPATestDB
connectiondrivername (default)
The JDBC driver class to use.
org.apache.derby.jdbc.ClientDriver
user (default)
The database user to login with.
derby
Save
Cancel