Pulsar Documentation

Release 3.1.0

Date: 17-Nov-2014 10:08

Table of Contents

1 Documentation Notes

2014-09-08 Pulsar 3.0 Preview
2014-09-25 Pulsar 3.0 Release
2014-10-09 Pulsar 3.0.1 Release
2014-10-13 Pulsar 3.0.2 Release
2014-10-20 Pulsar 3.0.3 Release
2014-10-21 Pulsar 3.0.4 Release
2014-10-29 Pulsar 3.0.5 Release
2014-11-13 Pulsar 3.1 Release

2 Introduction

Pulsar is both a platform and an architecture focused on web based systems. As a platform Pulsar contains a server runtime which can host a number of applications, built in and developed by third parties. The platform also contains a delivery system by which applications can be packaged, delivered and deployed to Pulsar instances. As an architecture Pulsar provides guidelines for the development of applications and provides tools to support the entire development process.

2.1 Brief history

The development of the Pulsar platform began in 1997 and version 1.5 was released in 1999. Pulsar 1.9 was released in 2006. The development of Pulsar 3.0 started in 2013 and constitutes a major redesign of the core framework.

2.1.1 Release history

Version	Release Date	Codename	Comments
1.2	2001-01-15		
1.5	2001-02-22	Galahad	
1.6	2001-08-22	Zoot	
1.7	2002-03-06	Bedevere	
1.8	2004-09-14	Arthur	
1.9	2006-04-10	Camelot	
2.0	2007-09-21	Excalibur	
2.1	2008-05-27	Excalibur	Presentation and SQL frameworks.
2.2	2009-03-03	Excalibur	JDK 6.
2.3	2010-05-05	Excalibur	Grouped overviews in presentation framework.
2.4	2011-11-02	Excalibur	
2.5	2011-12-11	Excalibur	SSO support.

Version	Release Date	Codename	Comments
2.6	2013-03-28	Excalibur	Support for UI levels.
3.0	2014-09	Merlin	New service oriented module system and dependency injection

3 Concepts

This section provides an introduction to and a definition of the concepts which are used in the rest of the Pulsar documentation.

3.1 Dictionary

3.1.1 Pulsar

Pulsar is a system architecture and a platform for running modules developed according to the architecture. Pulsar also provides development tools to support developers in the development of Pulsar modules.

Pulsar is more or less application agnostic but provides a lot of functionality specific to web centered applications. However, the core architecture of Pulsar can be used to develop any kind of Java application.

3.1.2 Modules

A Pulsar module is the typical modular building block. The Pulsar module can have import dependencies on packages and services and also export packages and services for use by other modules. Pulsar modules are packaged as jar-file artifacts. The Pulsar module concept is used both for development of the Pulsar platform and during application development using the Pulsar platform

3.1.3 Activators

Each module can have zero or one activators. The activator is a class that the module uses to interact with the Pulsar lifecycle management. Methods in the activator is called by Pulsar when the module is installed, upgraded, started or stopped.

3.1.4 Packages

A package in Pulsar is the same as a Java code package. Packages are versioned using the semantic versioning strategy.

The term package is also used to mean a packaged repository. See "Repository".

3.1.5 Services

Services are Java objects which are published by one module and possibly consumed by one or more other modules. Services are usually published on module start and unpublished during module shutdown.

3.1.6 Service notifications

Service notifications are asynchronous callbacks which are invoked by Pulsar when a service becomes available or unavailable, i.e. when it is published or unpublished by the providing module. These notifications are used to utilize dynamic runtime service dependencies.

3.1.7 Applications

Applications consists of one or more Pulsar modules. An application is a loose grouping of, usually, interdependent bundles. There are no strict rules regarding what modules should define an application. The application concept is just a way of organizing Pulsar modules in a hierarchy and create artifact names based on this hierarchy. The hierarchy may be used to create a composed deployment artifact containing multiple modules but the modules may also be packaged as separate artifacts.

3.1.8 Life cycles

Modules and services have life cycles within the Pulsar lifecycle which means that a module can be loaded, started, stopped and unloaded while Pulsar is running. Similarly services have a lifecycle within the module. Services may be published and unpublished while a module is running, but usually they are published when a module starts and are unpublished when a module is stopped.

3.1.9 Versioning

Modules and packages are versioned which means that they are assigned a version number. The version number of a module is visible in the file name of the module artifact. The version number of each package is used internally to determine that all dependencies between modules are satisfied.

Pulsar uses semantic versioning, which is a system of rules used to decide how the components of a version number should be incremented when the code in a package is changed or how a module version should be updated if package versions are changed.

3.1.10 Dependencies

Modules depend on functionality and resources in other modules. Dependencies are defined between packages and expressed as a version range for which the dependency is satisfied. Dependencies are handled transiently which means that the complete chain of dependencies will need to be satisfied for a consistent system state to be achieved.

3.1.11 Dependency injection

Dependency injection is a programming style which strives to make dependencies between code objects explicit. One of the benefits is that it makes code more testable. A very much simplified description is that all the dependencies (other objects) needed by an object should be provided as parameters when the object is created. Without some kind of library support for dependency injection the number of parameters that needs to be passed around can grow quite substantially. Therefore libraries and tools are often used to manage the process of dependency injection. Pulsar has adopted the Google Guice model of dependency injection and also uses Guice under the hood.

Pulsar provides dependency injection locally (local binding) within modules as well as between modules (service publish/consume).

The concept of dependency injection is well established within the software development community. See this Wikipedia article for further information.

3.1.12 Aliases

Services and modules must all have unique names to identify them internally for dependency and versioning purposes. However these names may be too cumbersome or internal to use in contexts where they are exposed to end users, for example in URLs. Therefore aliases may be used to shorten service and module names and hide internal implementation details.

3.1.13 Resources

Pulsar resources include all non Java code resources that might be packaged within a module. For example html templates, stylesheets, localization data, configuration files or other static data.

3.1.14 Configuration

Pulsar provides a configuration model that every Pulsar module can use. The configuration model is based on a default module configuration which is packaged with the module. When first installed in a Pulsar instance the default configuration is applied. Pulsar provides services and user interfaces through which the configurations of modules can be updated. The updated configurations are then persisted internally, between restarts, by the Pulsar instance.

3.1.15 Actions

An action is an event generated by a client. For example an HTTP request from a web browser to a specific page template provided by a specific module. The action is then defined by the template path and name. Other types of actions are calling a Pulsar metod from JavaScript or invoking a REST or Web Service.

3.1.16 Artifacts

An artifact is a binary file created in the packaging (distribution preparation) process. Artifacts created by Pulsar may contain one or more modules and their resources.

3.1.17 Repository

A repository is an indexed set of modules which can be searched by Pulsar to retrieve modules and all their dependencies. Multiple repositories can be used by one Pulsar instance and each repository may contain several different versions of each module. Repositories can be used to distribute modules both during development and in production systems.

A repository can reside on a HTTP server or a local disk as a number of jar-files and an index file. Repositories (index and jar-files) can also be packaged into files and are then given the file extension ".pulsar". Pulsar provides tools to create the index file and perform the packaging.

3.1.18 Testing

When referring to testing, this means running an automated test suite. Each module should have a suite of tests that validate it's functionality. When a test validate functionality inside of one module without other modules being involved (they might be mocked or just not needed), it is called a Unit Test. When a test validates a module API from the "outside" and involves other modules in the test, it is called an Integration Test.

4 Development reference

4.1 Module system

4.1.1 Introduction

Modules are the basic building block in the Pulsar architecture. A module is also the smallest deployment unit. Modules are then organized into layers or groups which are then composed into entire applications or systems. Pulsar defines the module but leaves the layering and composition open and to be guided by the specific requirements and architecture of the system.

Pulsar also combines modularity with dynamism which provides the capability to deliver and update modules in runtime, without the need for a restart of the application process.

Each module may publish services and consumes service, published by other modules. To keep the code syntax succinct and easy to understand Pulsar embraces the concept of Dependency Injection to deliver services used between modules. Dependency injection is also an enabler for better unit testing both for integration tests and unit tests.

4.1.2 Some benefits of a modular architecture

Structured modularity has enormous long term benefits both during system development, deployment and maintenance.

During system design it provides the ability to reason separately about well defined parts of the system without having to know the internals of each part. This provides the base for flexible system architecture which can be models and remodeled as the requirements change and the system evolves.

During development the boundaries between parts of the system become clear which makes it easier to reason about interfaces between modules and the responsibility of modules. Team responsibilities can be more easily assigned and work can be performed in parallell.

During deployment and maintenance the modular structure allows for flexible deployment and upgrades of newer versions or additional modules as the system evolves.

4.1.3 Modules as building blocks

Modules are the building blocks in Pulsar. Each module contains packages (defined in the Java code), published services (Java interfaces) and resources (any type of file).

Packages

The Java packages, and the Java classes belonging to these packages, are used to identify and limit what parts of the module code may be accessed by other modules. Pulsar uses a package name pattern to identify which packages should be exported for use in other modules. If a package contains a level called 'api' then it is automatically exported by Pulsar. Package dependencies is also used to determine transient module dependencies during installation and upgrades.

When a module is loaded in Pulsar the packages it exports are made available to other modules which require it's classes. Classes needs to be exported when they are used as parameters or return values in services i.e shared value objects. Packages dependencies doesn't need the module to be started and are removed when when a module is uninstalled.

Services

A service in Pulsar is basically a Java interface class. The service defines a set of Java methods which can be invoked by a service user. Services can be used internally and from other modules, using the same syntax. This makes it easy to create a service which one module and the extract it to a separate new module as part of a refactoring action. Services can only be published when a module is the started state. Services are even more dynamic than packages. In Pulsar services may be added by a module after it has been started. All services are automatically shut down when a module is stopped.

Resources

Resources are files that are used by the module like HTML files or configurations files. Resources are mostly accessed by Pulsar internally but may also be exposed and shared between modules. In most cases it is recommended to keep module resources private. If they should be available to other modules this can be achieved with more control through a published service API. Resources are not made available in Pulsar unless a module is started.

4.1.4 Module Versioning

Pulsar uses semantic versioning. For an introduction to semantic versioning please find more information at http://www.semver.org.

Modules

The module version is the version of the entire module and should be updated whenever anything in the module is changed. Module versions is a part of the filename used for module artifacts. Modules are versioned through the `@PulsarModule` annotation which placed the `package-info.java` file in the master package, together with the symbolic name (unique

identifier) of the module and other meta data which is later used by Pulsar. A Pulsar module requires exactly one `package-info.java` file with the `@PulsarModule` annotation to be present.

package-info.java

```
@PulsarModule(
        symbolicName ="symbolic.name",
        version="1.2.3",
        startLevel="10",
        presentationName = "Presentation name",
        description = "Longer description of the module"
        )
package ...;
import se.dse.pulsar.core.api.PulsarModule;
```

Optionally a "startLevel" attribute may be used. The start level determines the module start order and is used to roughly order the modules to allow them to publish their services in an order which corresponds to the module inter dependencies. Pulsar uses start levels 1 to 5 internally. Start levels 6 to 10 may be used for your modules.

Packages

Package are version through the standard Java mechanism for adding package meta data, the `package-info.java` file. Each api package must have this file and must be versioned according to the principe of semantic versioning. The package version is stored as meta data in the module artifact and used during dependency resolution.

package-info.java

```
@Version("3.5.0")
package com.example.api;
import aQute.bnd.annotation.Version;
```

Services

Services interfaces are versioned through the Java package they reside in. As a consequence all services interfaces in the same package are versioned together. It is therefore useful to separate different API aspects in sub packages to the main api package to version them independently.

4.1.5 The module activator

A module activator is a special Java class used to activate a module. Modules must have exactly one activator and it is defined by that it extends the class `PulsarActivator`. The only requirement is that the `init(ModuleContextBuilder)` method is implemented. The PulsarActivator is used to publish services, consume services and perform any initialization or cleanup needed during the starting or stopping of the module. Service publishing and consumption is handled through the `ModuleContextBuilder` interface which is provided as a parameter to the `init` method.

For details on the methods in `PulsarActivator` please see the JavaDoc.

The example below shows a complete activator which consumes the `MessageManager` service which is provided by Pulsar.

```
package com.example.module.alfa;
import se.dse.pulsar.core.api.ModuleContextBuilder;
import se.dse.pulsar.core.api.PulsarActivator;
import se.dse.pulsar.module.messagemanager.api.MessageManager;
public class AlfaActivator extends PulsarActivator {
    @Override
    public void init(ModuleContextBuilder i_contextBuilder) {
        i_contextBuilder.consume(MessageManager.class);
    }
}
```

4.1.6 Services and Dependency Injection

Service publication

A published service must be a Java interface. The publication can be backed by either a class or an object instance which implements the interface. To enable customized services where the behavior somehow is dependent on the consumer, a service factory can be used. See JavaDoc for details.

```
i_contextBuilder.publish(MyService.class).usingClass(MyServiceImpl
.class);   // publish using a class

MyOtherService l_otherService = new MyOtherService();
i_contextBuilder.publish(MyOtherService.class).usingInstance(
l_otherService);   // publish using an instance

PulsarServiceFactory l_myFactory = new MyFactory()
```

```
i_contextBuilder.publish(MySpecialService.class).usingFactory(
l_myFactory);  // publish using a factory
```

Service publishing and local binding must be uniquely resolvable. Therefore a module may not publish or bind locally the same service interface more than once. Sometimes multiple services of the same interface (variants) needs to be published, then each publication needs to be uniquely named using the `ModuleContextBuilder` at publication/bind time.

```
i_contextBuilder.publish(MyService.class).usingClass(MyServiceImpl
.class);                              // default implementation
i_contextBuilder.publish(MyService.class).usingClass(
MyServiceSecureImpl.class).named("secure");    // secured
implementation
```

Scopes

A service published with an implementation class always have a scope which determines when a new instance of the service should be created. The scope is determined by the implementation class and defined through annotations.

Scope	Implementation class annotation used	Description	Instantiation
Module	none	A new instance of the service is provided to each module which is using it. Each injection within a module will share the same service instance.	Once for each consuming module.
Singleton	@Singleton	A single instance is used during the entire lifecycle of the publishing module. All consumers share this instance.	Once when module starts (during injector creation).
Instance	-	When an object instance is published this instance is always used.	Manually before publication.
Custom	-	A PulsarServiceFactory can be used to create new instances depending on custom parameters. The factory may also decide wether consuming module or inject points should share service instances to receive separate ones.	Factory is instantiated manually and the factory then controls service instantiations.

Service consumption (injection)

Any service published by any Pulsar module can be consumed by any other module, including the publisher, through dependency injection. All services that a module wants to consume must be declared in the module activator.

```
i_contextBuilder.consume(MessageManager.class);
i_contextBuilder.consume(MyService.class);
```

Once the service is declared through `consume()` it may be injected in any constructor in any module class using dependency injection.

```
@Inject
public MyClass(MessageManager i_messageManager, MyService
i_myService) {...}
```

To inject a specific service variant use the `@Named` annotation.

```
@Inject
public MyClass(MessageManager i_messageManager, @Named("secure")
MyService i_myService) {...}
```

To inject all available instances of a service an `Iterable` is used at inject time.

```
@Inject
public MyClass(Iterable<MyService> i_myServices) {...}
```

Service notifications

Since services may be published or unpublished by their providing module at any time it is useful to be able to register a notification callback when consuming a service. Pulsar supports notification callbacks for injected services. A notification listener is registered with the injected service object and notifications are then provided by Pulsar.

To register a notification listener, first cast the service object to a `PulsarServiceNotifier` and then use `registerListener()` to add the listener.

```
private volatile boolean serviceInitalized = false; // volatile
since multiple threads may access it concurrently

@Inject
public MyClass(MyService i_myService) {
     PulsarServiceNotifier l_myServiceNotifier = (
PulsarServiceNotifier)i_myService;
    l_myServiceNotifier.registerListener(new
PulsarServiceNotifier.ServiceListener() {
            @Override
```

```
            public void onAvailable() {
                try {
                    i_myService.setup();
                    serviceInitalized = true;
                } catch (ServiceUnavailableException e) {
                    // Note: Service may become unavailable at any
time, so we need to be prepared to handle that.
                }
            }
            @Override
            public void onUnavailable() {
                serviceInitalized = false;
            }
    });
}

// ... more code using the serviceInitalized field ...
```

Some important notes on service notifications:

- All invocations to the listener methods are made using separate threads. The event receiver may preform any kind of work on these thread and there are no restrictions on the duration for which these events are processed. However only one event may be processed at a time. Pulsar will hold any subsequent notifications until the current event has finished processing.
- The `onAvailable()` method will be invoked directly if the service is available when the listener is registered. If the service is not available when the listener is registered `onAvailable()` will be invoked asynchronously when the service becomes available.
- Pulsar guarantees the invocation order between `onAvailable()` and `onUnavailable()`. I.e. when an `onAvailable()` event has been fired `onAvailable()` will only be invoked again if `onUnavailable()` has been invoked in between.
- If the service state changes, one or multiple times, during event processing, these changes will be dampened until the current processing is complete. I.e if an available-event is being processed, two concurrent unavailable and available-events will cancel each other out.
- Since services may become unavailable at any time there is no guarantee that service invocationsmade within the onAvailable() method won't throw a ServiceUnavailableException.

Listener unregistration is optional. Any remaining listeners will be automatically cleaned up when the Pulsar module is stopped.

Service invocation

When a service has been injected it's methods can be invoked as any other java object. However, since modules can come and go in runtime, as a result of modules being installed, uninstalled and upgraded there is no guarantee that a the backing service still will be available when invoked. To handle this dynamism Pulsar uses service proxies on injection. The proxy then invisibly handles the invocation according to the invocation strategy selected when the service was consumed.

Services may be consumed using three different strategies

Consume strategy	Description	Code example
Default	When a matching service is not available at invocation time a `ServiceUnavailableException` is thrown directly.	`i_contextBuilder.consume( MessageManager.class);`
Timeout	When a matching service is not available at invocation time the invocation will be halted until a matching service becomes available or a timeout expires.	`i_contextBuilder.consume( MessageManager.class) .invokeTimeout(5000 );// waits for service, max 5 seconds`
Block	When a matching service is not available at invocation time, block the invocation until a matching service becomes available.	`i_contextBuilder.consume( MessageManager.class) .invokeBlocking(); / / waits for service indefinitely`

As long as a service is available when an invocation is done Pulsar guarantees that the service will not be unloaded until an ongoing method invocation is completed. Even if the module providing the service is requested to stop during the ongoing invocation the release of the service will be delayed until the invocation is complete and the service is no longer in active use .

Local binding (injection)

Every Pulsar module may also use internal dependency injection where the bound objects can be injected in the same way as consumed services. Locally bound services are not available to other modules and can only be used within the module.

```
i_contextBuilder.bindLocal(MyService.class).usingClass(
MyServiceImpl.class);
```

Pulsar also support assisted injection through factory interfaces.

Important notes on Constructors and Dependency Injection

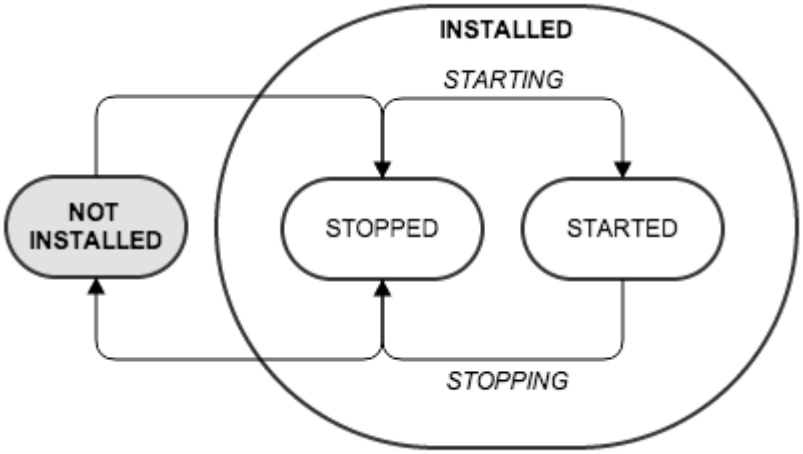
Avoid placing any method calls and logic in a class constructor To ensure that the code which uses dependency injection can be easily debugged, never do any real work in the objects constructor. Ideally the constructor should only assign `private final` variable with the injected constructor parameters. When an exception is thrown in a constructor during dependency injection this exception is handled by the dependency injection code and the complete stack trace is not displayed and debugging is much more difficult. Therefore the dependency injection and other types of initialization should be clearly separated.

A constructor in a class that uses dependency injection should look like the following example:

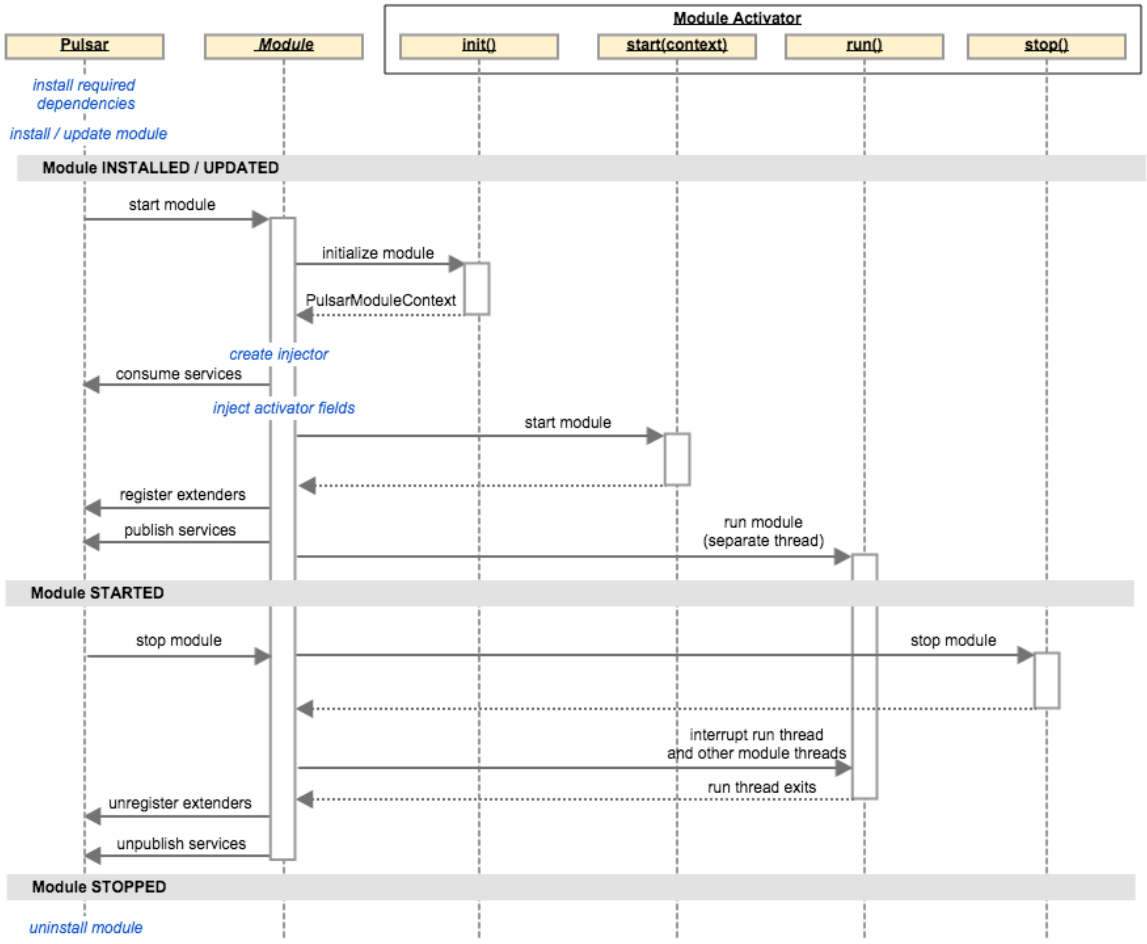
```
package com.example.good

import javax.inject.Inject;

public class GoodExample {

  private final ExampleService exampleService;
  private final OtherService otherService;

  @Inject
  public GoodExample(ExampleService i_exampleService, OtherService
i_otherService) {
    exampleService = i_exampleService;
    otherService = i_otherService;
    // do nothing more here!
  }

  ...
}
```

4.1.7 Module Life cycle

Pulsar defines the lifecycle of each module. The module lifecycle has two levels, deployment state and activation state. The deployment states are NOT INSTALLED and INSTALLED. And the activation states, which are sub states of the INSTALLED deployment state are STOPPED and STARTED. The activation state also has two transient modes which are STARTING and STOPPING.



The sequence diagram below describes the details of the module lifecycle and how the module and Pulsar interacts.



Lifecycle methods

Starting

Pulsar defines four lifecycle methods in a Module Activator, only one `init()` is mandatory to implement. The methods `init()` and `start()` are executed synchronously but on separate threads, taken from a thread pool. The `run()` method is executed on the same thread pool but in an asynchronous way which allows the implementor of the activator to hold on to the run-thread for any long running tasks.

To provide a consistent runtime environment Pulsar uses a module specific thread group to monitor all the threads used for lifecycle management, and all threads spawned from these. This also allows Pulsar to define the thread context class loader as the module class loader.

Stopping

Before invoking the `stop()` method Pulsar will shut down the dedicated thread pool for the module and then check for any active threads. If active threads are found these are passed, as an array parameter, to the `stop()`-method. The `stop()` method is expected to shut down the threads which are still active and release any other resources which otherwise risks to be left behind as leaks.

If the `stop()` method fails to shut down all active threads Pulsar will log a warning for each surviving thread since this is a possible resource leak. To prevent false warnings when execution services are used, parked threads are not flagged as leaks.

The `stop()` method is executed synchronously directly on the event thread and should never perform allocation of new resources. Its implementation should also be quick to execute and never block.

Module dependencies and Lifecyle

When a module is stopped it's services are also unpublished automatically. This means that other modules will not be able to access these services until the module publishing them is started again. Calling a service which is not available will throw a runtime exception of type `ServiceUnavailable`. See "Service invocation" for different options/strategies on invocation.

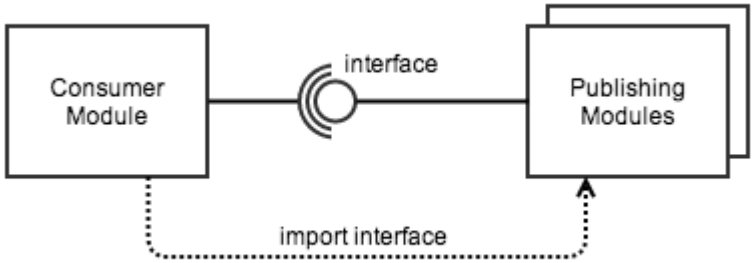
When a module is uninstalled all modules which import packages from that module will be stopped, since the classes in the packages imported are no longer available. This may result in a cascade of modules being stopped. To control the lifecycle behavior of dependent module it is necessary to have a carefully planned module structure. Undesirable dependencies can often be refactored into a structure where updates to one module will cause minimal disruption to others.

4.1.8 Central design patterns

Whiteboard

The whiteboard design pattern is central to service oriented architectures. Modules participating in the Whiteboard pattern typically consists of one listener module and zero or more modules publishing services. In a nutshell this pattern allows developers who wants to add functionality to the system, to do so simply by publishing services. Or in other words publish something on a board where it is then picked up by the system. The system then handles these service publications appropriately. In a dynamic service environment it is usually easier to publish a service compared to consuming one. This pattern also loosens the coupling between modules, which is good.

Pulsar supports the Whiteboard pattern through it's publish consume model. The publishing part simply published the service according to the interface specified by the consumer module. The consumer consumes the same interface and injects an iterable over the same interface to be able to find all the service instances.



Extender

The extender pattern describes how a module can act as an extender of another module. In this context extending means adding functionality to. An extender needs to know when an extendable module becomes available or is removed from the system, therefore extenders are implemented as listeners which can receive events about other modules. The extender pattern complements the whiteboard pattern since it can act on the configuration or resources of a module without the module having to publish any services. An extender may even publish services on behalf of a module it extends.

In Pulsar extenders are classes registered through the `PulsarModuleContextBuilder` using the `extendUsing()` method.

NOTE: If multiple extenders are used it is advisable not to have dependencies between these extenders. The order in which the different extenders are notified about a module becoming available or disappearing is currently undefined.

4.1.9 Notes on creating a dynamic modular architecture

The first step towards creating a modular architecture is to decide on which modules are needed. These general rules may help:

1. Avoid complex dependencies by never allowing circular dependencies between modules. The module graph should be directed and not contain any loops.
2. Identify common value objects and refactor them to separate modules which can be reused throughout the system.
3. When designing a modules public API it is essential to keep it simple. Avoid API state where possible.
4. Keep the value objects, parameters and return values, simple and immutable where possible.
5. Module APIs should be fairly stable (forward compatible) to avoid unnecessary downstream adaption work.

4.2 Development environment

4.2.1 Notes on the Pulsar development environment

The development environment is the set of directories, files, tools and processes used to develop modules for the Pulsar system. Pulsar dictates some core parts of the directory structure. But most of the development environment is up to the developer or architect to decide up on. This documentation will describe what is mandatory and what the best practices are for setting up a development environment for Pulsar module development.

There are many different integrated development editor (IDEs) and Pulsar module development should be possible using any of them. Here the general case is described, using the command line for scripts and any code editor for file editing.

4.2.2 Setting up an application development environment

Setting up Pulsar

Pulsar is distributed, by DSE, in four required components/files:

Component	Distribution file	Description
Launcher	**pulsar.launcher-***version*.**jar**	The launcher is an executable jar file.

Component	Distribution file	Description
Pulsar Core Modules	**pulsar-modules-***version*.**pulsar**	A package containing the core modules needed to run Pulsar.
Pulsar Apps	**pulsar-apps-***version*.**pulsar**	Package containing essential applications which are used to monitor, configure and manage the Pulsar instance.
Endorsed Libraries	**pulsar-endorsed-***version*.**pulsar**	Third party libraries used by Pulsar Core Modules and Pulsar Apps.

4.2.3 Directory structure and essential files

The directory structure used during application development, using a deployed version of the pulsar runtime. Directories and files marked with red are created in runtime or manually added and should be excluded from version control. Files and directories marked in red are generated as part of the build process or created in runtime, these should never be put under version control.

Directories and files					Description
<pulsar>/					Pulsar installation directory.
	.pulsar-runtime/				Internal runtime directory. Contains the runtime cache and internal persistence. Created and managed by Launcher.
	apps/				Application modules.
		<modulegroup>/...			A series of application module directories. Modules can be ordered into an arbitrary directory structure where every leaf contains a module.
		<module>/			A module being developed.
			conf/		Module configuration files.
				<module>.conf	Module main configuration file. This file contains the configuration specification of the module, i.e. all allowed configuration parameters, their descriptions and value domains. The configuration is automatically picked up by the Configuration Manager when the module is installed and presented through the Pulsar Configuration Editor. See Configuration Manager and Configuration for configuration features and syntax.
				permissions.xml	Module permission configuration. See Permission Manager for features and syntax.

				resources.xml	Module resource configuration. See Permission Manager for features and syntax.
			depend/		Dependencies needed for compilation, testing and distribution.
			dist/		Output directory for packaged distributable versions of the module.
			docs/		Generated JavaDoc.
			lib/		
				exported/	Libraries (jar) which should be exported as part if the modules public API. These libraries will become available for import from other modules.
				internal/	Libraries (jar) which are internal to the module. These libraries will not be accessible to other modules.
				native/	Native libraries (dll, so, dyl) to be used internally.
			resources/		
				static/	Static resources.
				stylesheets/	Stylesheet resources (XSL).
				templates/	Templates with dynamic content.
			script/		
				build.xml	Module build file. Imports build.macro.xml.
			src/		Java source code for module
			src-gen/		Generated module specific Java source code.
			src-test/		Java tests source code for module.
	conf/				
		logback.xml			Can be included in version control to keep a shared default between developers on the project.
	data/				

					The data directory is used to provide a data area for each Pulsar module where module specific runtime data may be stored and accessed. Typically this is used for module specific resource files which may be updated in runtime. Resource files in the data area will shadow any resources packaged in the module artifact.
	dist/				Destination for application packages and local development repository.
		repository.xml			Development repository index. Includes all modules which are developed within the project.
	lib/				Pulsar libraries and tools needed to build and package modules and applications. **Note: Dynamic compile time dependency resolution will be available in a future versions of Pulsar. This will eliminate the need for most of the files in the lib structure and further ease module development.**
		compile/			Compile time libraries, including libraries to use for logging (SLF4J) and the Pulsar Module APIs. The contents of this directory should be be available on the classpath when compiling modules.
		dist/			Libraries needed by the Pulsar scripts to package modules.
		test/			Libraries needed to run test cases. Should be available on the test classpath.
		tools/			Tool libraries needed by the build scripts.
	logs/				Runtime logs.
	script/				Build and package scripts. Scripts are written for Ant.
		build.macro.xml			Macro definitions needed by module build scripts.
		build.properties			Global build properties.
		build.xml			Master build script. Used to compile/test/dist all modules in the project.
		package.macro.xml			Macro definitions used by the package script.
		package.xml			Master package script. Used to compose modules into packages.

	pulsar.launcher-*version*.jar				Pulsar Launcher.
	pulsar-modules-*version*.pulsar				Pulsar Core Modules.
	pulsar-apps-*version*.pulsar				Pulsar Apps.
	pulsar-endorsed-*version*.pulsar				Pulsar endorsed libraries.

Libraries

Pulsar provides a number of core modules. Most of these modules have interfaces which provide functionality for an application developer. During compile time these interfaces are provided through the packaged module files (jar file) which are distributed together with Pulsar.

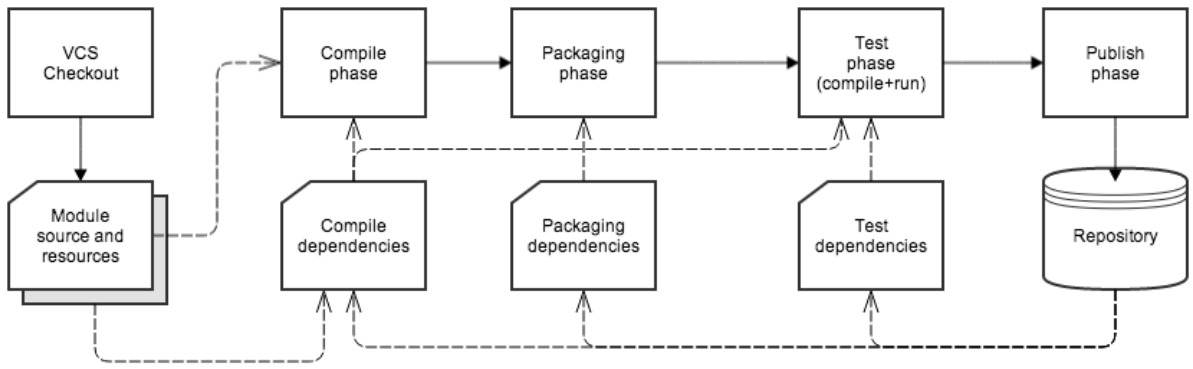
The Pulsar core modules also requires a set of third party libraries to work. These third party libraries are called endorsed libraries and are provided in a separate package within the Pulsar distribution. Most of these libraries are only used internally but a handful are also required for application development. For example the Google Guice libraries for dependency injection are required whenever a Pulsar module is compiled, also we recommend using the SLF4J libraries for logging.

Native libraries

Native libraries also needs to be configured in `<module>/conf/manifest.conf`.

4.3 Build and Continuous Integration

4.3.1 Building a Pulsar module



A Pulsar module is built through the phases of compilation and packaging (also referred to as " dist", as in "distribution"). Both these phases have dependencies on other modules. For example a module has a compile time dependency on all the modules which APIs are used. Since the Pulsar Core Module is used by all other modules this is required during compile time. All modules have a "dist" dependency on the Pulsar Packager Module since it is needed for packaging.

The dependencies that are required for each phase can be found in the current project if we are developing multiple modules together or through a repository. The repository may be local (a repository XML file), a ".pulsar" file or a ".pulsarlink" file containing a URL pointing to the location of the repository. See Repositories.

Pulsar provides scripts for building modules. These scripts also handle dependencies and fetches the required set of dependencies for each phase.

Build scripts

The standard Pulsar build scripts are written for the Ant tool. Each module build script `<module>/script/build.xml` includes the master script `script/build.macro.xml` and uses it's macro definitions to specify the module build.

Example of a simple module build script:

```
<project name="my-module" default="dist" basedir="..">
    <import file="../../../script/build.macro.xml"/>
    <pulsar-module
            dir="."
            pulsar.dir="../../"
            />
</project>
```

The `pulsar-module` entry is expanded (by Ant macros) into a set of tasks corresponding to the different build phases. The following attributes may be passed to the `pulsar-module` macro:

Attribute	Default	Description
pulsar.dir		A reference to the Pulsar directory, which usually is the root of the project.
dir		A reference to the root directory of the current module.
dependencies-compile		A comma separated list of the other modules needed to compile this module.
dependencies-compile-test		A comma separated list of the other modules needed to compile the tests in this module.

Attribute	Default	Description
dependencies-test		A comma separated list of the other modules needed to run the tests in this module.
dependencies-dist		A comma separated list of the other modules needed package (dist) this module.
generate-param-signatures	true	Turn on or off the generation of Param Manager signatures for Pulsar Methods.
moveinternalclasses	true	Hide internal classes (non api) within JAR when creating a dist.
generate-javadoc	true	Generate javadoc for apis.
clean-src-gen	true	Delete the `src-gen` directory in clean phase.

The tasks generated by the Pulsar build macro are:

Task	Description
clean	Deletes all generated or downloaded data, classes, javadoc, test reports, dependencies and module packages.
compile	Compiles the module source into class files and creates meta-data for the methods annotated with @ PulsarMethod.
compile-test	Compiles the module test sources.
dist	Builds a packaged version of the module (calls "compile" if needed)
test	Runs the tests (calls "compile" and "compile-test" if needed)

Other Ant scripts may be added as required, for example we recommend having a master build script which can perform tasks for all modules in a project.

4.3.2 Testing a module

Testing of a module adds two more phases apart from the build phases (compilation and packaging), these are test compilation and test running. These two phases also have their own dependencies. To compile the tests JUnit and the Pulsar Test Module are usually required. The running of the tests might have further dependencies on modules which are used as part of an integration test.

4.3.3 Packaging several modules into package file

Modules may be arbitrarily grouped and packaged into package files to be used for deployment. The package files have a ".pulsar" suffix and are zipped archives containing both the individual module jar files and a repository index listing the contents of the deployment file.

Pulsar provides a packaging macro (`script/package.macro.xml`) which can be used to create these repository files.

4.3.4 Continuous builds

Pulsar modules can easily be build using a continuous build system such as Jenkins or Bamboo . The provided Ant scripts can be configured to be invoked by the continuous build engine and the artifacts (repositories and package files) can be collected in the dist directories.

4.3.5 Pushing changes from a continuous build system to a module repository

Best practice is to automatically publish successfully built modules to a module repository where they are available for installation and upgrades. Different repositories can be set up for different versions of the modules. Usually separate repositories are set up at least for a development branch and a master branch.

Pulsar provides scripts for adding a set of module artifacts to a repository and update the repository index. See Repositories for more information.

4.4 Pulsar Core Modules

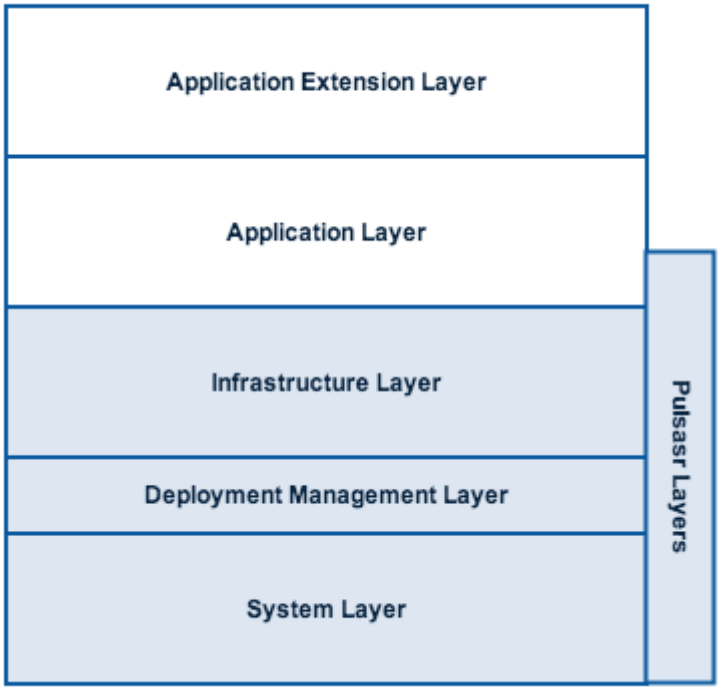
4.4.1 Introduction

The Pulsar Core modules contains the essential Pulsar packages and services. Together they make up the Pulsar framework and platform. The Core modules are packaged in the `pulsar-modules` package and are contained in the system, deployment management and infrastructure layers, as described below. Generally the Core modules have no end user interfaces, only APIs designed to be used through packages and services.

The documentation of the Core Modules consists of this reference document together with the JavaDoc published for each module. The purpose of the reference documentation is to give an overview of each module and to document configuration or usages which is not apparent in the API JavaDoc.

4.4.2 Architectural layers

The Pulsar architecture is layered into a number of different sections, where each section is responsible for it's functional domain. The lower layers contains the internals of Pulsar and should be more static than the upper layers which will change whenever applications are developed.



System Layer

The system layer contains the Pulsar Launcher and core libraries needed to bootstrap the platform. The primary focus of the system layer is to set up a minimalistic environment which is being able to load and start the deployment management, which is a module it self. Updates to the system layer should be very rare.

Deployment Management Layer

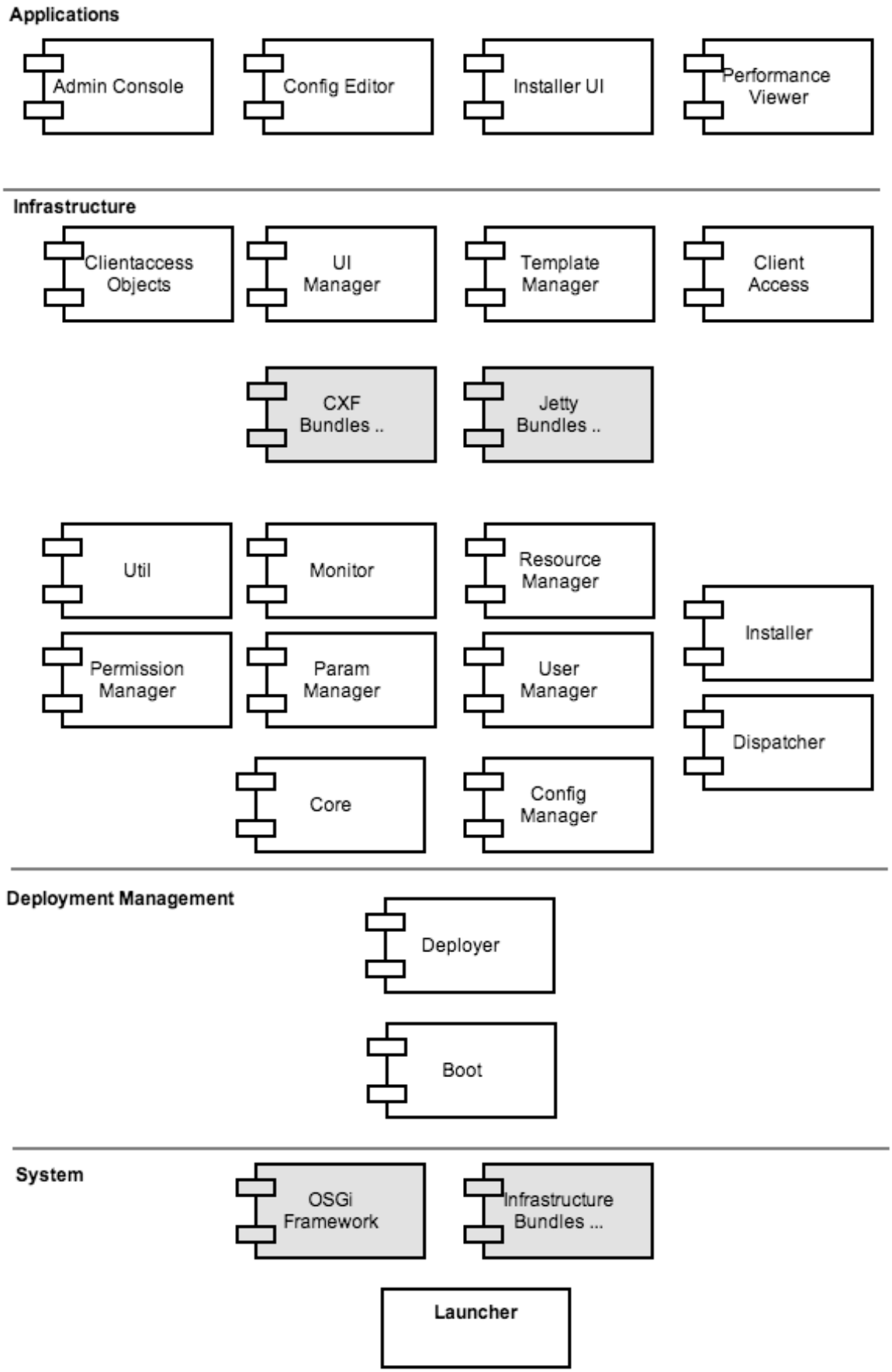
The deployment management layer is responsible for loading all other modules, either from local packages or repositories. The deployment management layer is also responsible for finding and loading any transitively required resources. When Pulsar is first installed the deployment layer initializes the instance according to a built in profile. Profiles can be used to pick different modules from the Core modules to build a platform with or without a specific capability, for example a developer instance or an instance with support for web services and rest services. For more information about available profiles see Installation and upgrades.

Infrastructure Layer

The infrastructure layer contains the bulk of the Pulsar API packages and services. The infrastructure modules works together to provide the framework services which can be used by application module developers.

4.4.3 Core Modules

Pulsar provides a number of Core modules which provides the core Pulsar functionality. Each Core module belongs to one of the architectural layers.



4.4.4 Configuration Manager

Introduction

The Configuration Manager module provides functionality for persisting and managing configurations for modules. Each module may choose to keep it's configuration in Configuration Manager which then provides the module with a set of configuration related functionality.

Features

Accessing the configuration programmatically

The configuration is delivered to the module as a `Config` object. The `Config` is provided as a service from Configuration Manager and may be retrieved in the module activator. The basic use case involves consuming the configuration and making it available to the rest of the module, by dependency injection:

```
public class MyModuleActivator extends PulsarActivator {
    @Override
    public void init(ModuleContextBuilder i_moduleContextBuilder)
{
        i_moduleContextBuilder.consume(Config.class);
    }
}
```

In the case where the activator needs to act on the configuration information during activation, for example to decide which services should be published och consumed, this is achieved by looking up the configuration service directly in the activator:

```
public class MyModuleActivator extends PulsarActivator {

    private Config config;
    @Override
    public void init(ModuleContextBuilder i_moduleContextBuilder)
{
        config = lookupService(Config.class);
        // also make config available through dependency injection
        i_moduleContextBuilder.bindLocal(Config.class).
usingInstance(config);
    }
}
```

Configuration templates

The Configuration Manager looks for `config/module_symbolic_name.conf` whenever a module is loaded into Pulsar. If the template file is present it is loaded by Configuration Manager .

Example of a configuration template:

```
settingA = DEFAULT VALUE
#description: A description of the setting and when it should be
used and if it may be changed in runtime.

settingB = DEFAULT
#description: A setting which is presented as a drop-down with a
limited set of values in the editor.
#values: DEFAULT,HIGH,MIDDLE,LOW

settingC = false
#description: A setting which is presented as a checkbox in the
editor.
```

Web based configuration editing

The parameters defined by the configuration template is made available for viewing and editing through the Configuration Editor application. The Configuration Editor is part of the Admin Console.

Import export

Configurations can be imported and exported. The file format is a plain text format based on the Apache Configuration format.

Live updates

When a configuration for a module is updated in the web user interface, or imported, the module is automatically restarted and provided with the new functionality.

Configuration

The Configuration Manager itself currently doesn't have any configuration parameters.

4.4.5 Database Access

Introduction

The Database Access module is used to interact with any JDBC accessible database. The Database Access module also handles connection pooling, caching and provides an easy to use interface for accessing data in a database. Database Access handles any number of concurrent pool configurations. The pools can also be shared between several Pulsar modules.

Configuration

To configure a database connection copy the following Database-config block into the module configuration file. Enter the values directly in the copied configuration file, or use the ConfigEditor to change the values.

Configuration file

```
<Database DbPoolName>
    DbURL = jdbc:...
    #description: A database connection URL is a string that your
DBMS JDBC driver uses to connect to a database. It can contain
information such as where to search for the database, the name of
the database to connect to, and configuration properties. The
exact syntax of a database connection URL is specified by your
DBMS.
    DriverClassName =
    #description: The JDBC driver to use for this database
connection.
    User =
    #description: User name to use to connect to the database.
    Password =
    #description: The users password to use to connect to the
database.
    InitialConnections = 2
    #description: The number of initial connection to create at
startup. The pool will always try to have this number connections
available.
    MaxConnections = 10
    #description: Max number of connection that the pool will
create.
    UseCache = false
    #description: Specify if caching will be used. If this
parameter is set to false, the parameters TableStatusHandler and
CheckType is ignored.
    TableStatusHandler =
    #description: A class that implements the interface
TableStatusHandler that returns a status string for a table.
Pulsar bundles two handlers which can be used: MySQL -
se.dse.pulsar.module.databaseaccess.cache.handlers.MySQLTableStatu
```

```
sHandler and for Oracle -
se.dse.pulsar.module.databaseaccess.cache.handlers.OracleTableStat
usHandler
    CheckType = Before [Before, Background]
    #description: Specify when the table status handler will check
if a table has been updated. Before = The check will be done
before every SELECT-statement. Background = The check will be done
in the background.
    InitStatement =
    #description: One or more statement separated with semi colon
which will be executed every time a new connection is created.
</Database>
```

Invoke the `processConfig` method to make the Database Access module process the configuration.

```
public class MyModule extends PulsarActivator {
    private Config config;

    @Override
    public void init(ModuleContextBuilder i_contextBuilder) {
        config = lookupService(Config.class);
    }

    @Inject
    DatabaseAccess databaseAccess;
    @Override
    public void run() {
        try {
            databaseAccess.processConfig(config);
        } catch (Exception e) {
            // Log problem with config file
        }
    }
}
```

Using a database pool

To use a configured Database a reference to a `DatabaseAccessPool` is needed. To get the pool reference, make a *named consume*, where the name is the same name as in the config.

```
public class MyModule extends PulsarActivator {
    private Config config;
    @Override
    public void init(ModuleContextBuilder i_contextBuilder) {
```

```
        i_contextBuilder.consume(DatabaseAccessPool.class).named("
PoolName");
    }
}
```

To get the actual referens to the `DatabaseAccessPool`, use a *named inject*.

```
public class MyModuleServiceImpl implements MyModuleService {
    private DatabaseAccessPool databaseAccessPool;
    @Inject
    public MyModuleServiceImpl(@Named("PoolName")
DatabaseAccessPool i_databaseAccessPool) {
            databaseAccessPool = i_databaseAccessPool;
    }
```

Adding JDBC drivers

Any JDBC compliant driver may be used with Database Access. To add a specific JDBC driver which is not already packaged for Pulsar the easiest way is to create a new module and put the driver jar in the `lib/exported` directory. The driver is selected through the connection URL specified by the `DbURL` configuration parameter.

4.4.6 Frameworks

Introduction

The Frameworks module provide standard functionality used to present information from a database in a user interface.

Configuration and database tables are used to define views which can be presented in overview mode (a table of data) or one row (one post) mode. Rules stored in configuration files are used to define the columns of the views and the frameworks allows the uses to dynamically configure new column sets and data filters. The setup for a dynamically configured view can also be stored and retrieved at a later time.

The Frameworks module handles the interaction with the database according to the configuration supplied. Usually a Java class is created for each data view to be set up. Data which can be used to render a data view is returned as XML which can be presented in HTMl by applying XSLT.

Features

Sorting

Views may be sorted descending or ascending based on data in one of the columns.

Groupings

Data may be grouped according based ont the leftmost n columns. The rows are then collapsed into groups according to their values int the group columns.

Column selections

Columns can be dynamically selected from the configured set of columns. A column may be marked as locked which means that it's viewing status (shown or hidden) may not be changed by the user, for the specific view.

Functions on columns

Functions, such as sum and average may be applied to columns.

Filtering

Columns may have filters which acts on the rows. Filters may contain expressions, for example to select only rows with a value within a certain range.

Formatting and links

Row values may be formatted according to custom rules and types. Links, units and custom formats can be added.

Modes and functions

Modes are used to create custom views for different user interface modes, for example a mobile mode may contain fewer columns for a specific view than the desktop mode of the same view. Functional flags are similar but used to indicate that a certain feature is enabled or not enabled. Functional flags are often used to enable special columns which are dependent on a certain function/feature being enabled.

Configuration

Configuration of the overviews can be done programmatically or in configuration files. The configuration is done in three levels where each level inherits the config from the level above. The main principle is that if nothing has been told in a lower level, the config from the level above is intact. Only changes need to be configured.

Default configuration

The default configuration is placed in a config file call `systemdefault.conf`. The syntax of this file:

```
<System>
    <Module Aaa>
        columnDefinitions = \
            { flag:_flag1_ flag:_flag2_ ... mode:_mode1_ ... }_
COLUMN_1_ID_[ _attribute1_:_attribute2_=_value2_: ... ],    \
            { _flags_ _modes_ }_COLUMN_2_ID_[ _attributes_],    \
            ...
            { _flags_ _modes_ }_COLUMN_N_ID_[ _attributes_]
    </Module>
    <Module Bbb>
    ...
    </Module>
...
</System>
```

Custom configuration

User configuration

Configurations of the overviews can be done programmatically or via configuration files. The configuration files is contained in the LEB/conf/modules folder. Configurations are done in three levels where the next level inherits from the previous one if no override is defined. These three levels are system settings, system override and user overrides. The basic principle is that if the overrides are empty the system settings are applied.

System settings

Every data view needs a configuration where a complete list of available columns are specified. This list is also the system default configuration.

Syntax for system settings:

```
<System>
    <Module Aaa>
        columnDefinitions = \
            { flag:_flag1_ flag:_flag2_ ... mode:_mode1_ ... }_
COLUMN_1_ID_[ _attribute1_:_attribute2_=_value2_: ... ],    \
            { _flags_ _modes_ }_COLUMN_2_ID_[ _attributes_],    \
            ...
            { _flags_ _modes_ }_COLUMN_N_ID_[ _attributes_]
    </Module>
    <Module Bbb>
    ...
    </Module>
...
</System>
```

Column configuration

Each column is configured in three steps:

1. First the filter flags are specified as a space separated string within curly braces. Usually these are flags which indicates for which functions and modes the columns should be available. The order of the modes and flags doesn't matter.
2. Next the column ID is specified. This string ID must match the column definition in the source code for the data view. Note that a column may defined multiple times, for example if it should be available for different modes with different settings.
3. Last comes a set of attributes which are to be applied to the column. The set is string separated with colon and surrounded by square braces. Ordering is irrelevant. Note, the square braces are mandatory, even if no attributes are specified.

1. Flags

Flags can be used to dynamically (per instance) decide if a column should be available or not. The flags are defined in the filter part of the data views (source code) and are therefore data view dependent.

All flags are specified with the `flag:` prefix. The flag value can be set in a HTML-template and may then affect how data is presented. The flags may also be set programmatically or through the request parameters. Flags may be specified directly or using negation, for example `flag:!integrationEnabled` will present the column if the flag `integrationEnabled` is set to `false` when the data view is rendered.

2. Modes

Modes are used to adapt the columns for different modes of presentation. Modes are specified with the `mode:` prefix. The set of available modes are defined in the data view source code and may be specified entirely by the application. If no mode flag is present the column configuration is applied for all modes.

3. Attributes

Attributes are used to further control the presentation of the column, when it is selected for presentation (by flags and modes). The following attributes are supported:

Attribut	Värden	Beskrivning
`hidden`		Defined the column as hidden.
`locked`		Define the column as locked. This means that a overriding level may not change the value of the hidden attribute. I.e. if a column is hidden and locked it may an overriding configuration may not make it unhidden. However a column may be `unlocked` by a overriding level.
`unlocked`		Unlocks a column if it was locked in a previous configuration. When unlocked the column may be hidden or unhidden again.
`filter=`	*filter text*	Defines a default filter for a column. May be specified and overridden on all levels.
`width=`	*pixel width*	Defines the presentation with of the column. May be specified and overridden on all levels.

`sort=`	`ASCENDING`
	`DESCENDING`
	`NONE`
		Specifies sorting. The value `NONE` is used to revoke an inherited sort order.

`filtertype=`	`TEXT`
	`LIST`
	`NONE`
		Type of filter.

`groupfunction=`	`SUM` `NONE`	Type of grouping function.
`showlink=`	`TRUE` `FALSE`	Specifies wether links from the presented values should be available or not.

Example of a system configuration:

```
<Module PU>
        columnDefinitions = \
            COL1[width=80], \
            COL2[width=160], \
            {mode:PORTAL}COL3[locked:hidden:filtertype=LIST], \
            {mode:PORTAL}COL4[width=160], \
            {mode:PORTAL}COL5[], \
            {mode:PORTAL}COL6[filtertype=LIST], \
            {flag:specialFunc}COL7[locked:hidden:width=40:
filtertype=LIST], \
            {flag:specialFunc}COL9[locked:hidden:filtertype=LIST],
\
            {flag:specialFunc}COL8[], \
            {flag:specialFunc mode:PORTAL}NIV2[width=80], \
```

```
            COLX[width=66], \
            {mode:PORTAL}COLY[groupfunction=SUM], \
            {mode:PORTAL}COLZ[groupfunction=SUM]
    </Module>
```

System overrides

System overrides are usually used to configure a system for different instances. For example different clients may want a different default setup.

When flags are used in the column definitions, all flags in the system configuration is checked first, then the system override flags are checked. For the column to be enabled both sets of flags needs to be enabled.

User overrides

Each user may be allowed to customize their data views. This is most commonly used to select columns and their order in each data view, for each available mode.

4.4.7 Message Manager

Introduction

Message Manager is a module that is used to send mail or SMS via Pulsar in a synchronous or asynchronous fashion.

Other than sender, reciever, subject and message, Message Manager include the ability to send other types of data related to the message such as:

- cc (Carbon copy)
- bcc (Blind carbon copy)
- Multipart

Features

Synchronized and asynchronized messages

Messages can be sent synchronously or asynchronously. Asynchronous messages are stored in a send queue and are then dispatched by a separate thread after the send method has returned to the caller. When a synchronous method is used the send method will block until the message has been dispatched or the dispatch failed.

Every method that include sending a message gives feedback on transport status. The feedback tells if the message could be sent or not.

Broadcasting messages

Message Manager has the ability to send messages or SMS to a multiple of receivers. Broadcasting can be done in a synchronous or asynchronous fashion. Broadcasting include the ability to send cc (Carbon copy) or bcc (Blind carbon copy).

Multipart

If the message need additional data related to the message a multipart containing this data can be appended and sent with the message.

Status

Message Manager contain the ability to retrieve the current status of messages that has been sent or not sent. The data that is checked are

- Successfully sent mails
- Not successfully sent mails
- Successfully sent SMS
- Not successfully sent SMS
- Current queue of mails and SMS to be sent

Configuration

To send messages with Message Manager it is required to configure servers, like a SMTP-server for sending email or a gateway service for sending SMS (text messages).

4.4.8 Permission Manager

Introduction

Permission Manager handles permissions for all resources (templates, methods or custom paths) in Pulsar. Permission Manager provides an API to test if a user has access to a specified resource.

The permission information is separated into resource groups and grants. Resource groups are collections of resources. Each resource group may contain any number of resources of different types. Grants then define the link between users or groups of users and the resource groups. Users and groups are handled by User Manager.

Permissions can be given to a certain user or user group. A resource in Pulsar module can be an HTML-template, a method or a static resource, such as an image.

Permissions are inherited, for example if a group is assigned a permission all the users and groups belonging to this group are also given the assigned permission.

Permission Manager is used internally by Pulsar to check which users have access to which templates, methods or other resources. Developers using Pulsar may also create custom path based permission hierarchies which can then be enforced using the same permission framework.

Usage

The permission are specified in two XML-files that is place in the module conf-folder:

resources.xml

Defines the resource groups.

Note: '*' matches any path element and '**' matches any set of path elements.

```
<resources>
    <resourcegroup name="alfa">
        <template path="public/**"/>
        <method path="com.corp.public.UserFeedback.postFeedback"/>
    </resourcegroup>
    <resourcegroup name="beta">
        <template path="admininterface/**"/>
        <method path="com.corp.admin.AdminModule.shutdownService"/
>
    </resourcegroup>
</resources>
```

permissions.xml

Defines the grants that link users and groups with the resource groups.

```
<permissions>
    <grant>
        <resourcegroup name="alfa"/>
        <group plugin="pulsar" id="everyone"/>
    </grant>
    <grant>
        <resourcegroup name="beta"/>
        <group plugin="myPlugin" id="Administrators"/>
        <user plugin="myPlugin" id="adminGuest"/>
    </grant>
</permissions>
```

Features

General mandatory permission control

Permission Manager checks all access to resources provided by Pulsar modules. By default no access is allowed and all exceptions needs to be configured in `resources.xml` and `permissions.xml`.

Custom permission control

Permission Manager also offers a plugin model for modules wishing to add custom permission checks. When a module implements the `PermissionControl` interface and publishes it Permission Manager will invoke the `check` method for all requests to resources belonging to this module.

> Currently at most one `PermissionControl` service may be published for each module, if more than one services with this interface are published the result is undefined.

Custom resource paths

Custom paths can be used to assign permissions to virtual resources within an application. For example, a custom permission hierarchy could be used to assign update or delete privileges in an object type hierarchy.

```
<resources>
    <resourcegroup name="alfa">
        <custom path="/objects/*/update"/>
    </resourcegroup>
    <resourcegroup name="beta">
        <custom path="/objects/*/delete"/>
    </resourcegroup>
</resources>
```

The effective permissions can then be checked programmatically using either permission tags or by calling `PermissionManager.checkModulePermission(...)`.

Permission tags

Permission tags can be used in templates to include content based on a permission check. See Template Manager.

Configuration

Permission manager uses caching of templates to increase performance. The cache invalidation timeout and clean interval can be configured.

4.4.9 Search modules

Introduction

The search modules provides a search engine for indexing and searching together with helper modules to index content from documents on disc and data from a database.

The search modules consists of three main modules:

- Engine
- DBScanner
- FileScanner

Engine

The Engine contains the search engine that is responsible for indexing and searching.

Indexing

To be able to put documents into the index you need an `Indexer`.

```
Indexer l_indexer = SearchEngine.getIndexer("indexName", "sourceId
");
Map<String, Object> l_map = new HashMap<>();
l_map.put("name", "John Smith");
l_indexer.put("id", "type", l_document);
```

Searching

Searching in an index is done by a `Searcher`.

```
Searcher l_searcher = SearchEngine.getSearcher("indexName");
Map<String, String> l_map = new HashMap<>();
l_map.put("name", "John*");
SearchResponse l_response = l_searcher.search("type", l_map, 10);
```

DBScanner

The DBScanner is used index data from a Database. Either you configure the module it self or you provide the config programmatically, as in the example below.

Configuration

```
<DbSearch>
    indexName = indexName
    databasPoolName = db
    sql.1 = SELECT id, name, age FROM persons
</DbSearch>
```

```
public void initIndex() {
    Config l_dbSearchConfig = searchConfig.getParameters("DbSearch
", "");
    dbScanner.start(l_dbSearchConfig);
}
```

FileScanner

The FileScanner is used to index files on a file system. In the same manner as the `DBScanner` the module can be configured by it self or getting a config programmatically.

Configuration

```
<FileSearch>
    indexName = indexName
    dir.1 = c:\documents
</FileSearch>
```

```
public void initIndex() {
    Config l_fileSearchConfig = searchConfig.getParameters("
FileSearch", "");
    fileScanner.start(l_fileSearchConfig);
}
```

The content of the files are extracted by the `FileExtractor` module.

FileExtractor

The FileExtractor is a helper module for FileScanner, but it can be used separately to extract content from files.

4.4.10 Session Manager

Introduction

Session Manager is responsible for the state of user sessions. A session is usually initiated when a client (browser) instance first connects to the system. Sessions are then tied to a user through an authentication step and then lives until the user disconnect, logs out or the session times out. Web sessions are tracked through cookies.

Features

Get the current session

Use the method `getCurrentSession` to get a `Session` object representing the current Session. The current session is determined through a thread local context. This means that sessions are assigned to the request thread servicing a certain browser request. Other threads may also have sessions assigned to them programatically.

Connect a user to a session

To connect a user to a session, typically in a log in method, you use the method `setUser`.

Time limit of a session

Time limits can be set on a session in order for it to expire after a given time of inactivity. This can be useful if the session contain steps with sensitive information.

Save and retrieve data from/to the session

The `Session` object has methods (`putValue`, `getValue` and `removeValue`) to connect data to the Session. The data is persistent as long as the session i alive. When the session is removed, the data connected to the sessions if removed.

Automatic data conversion for session data on upgrades

Since modules using the session to store data may be upgraded at any time the implementation of the value object classes may also change. Session manager can therefore automatically convert any session objects which are retrieved by an updated version of a module.

Configuration

Sessions will time out after a configured time of inactivity.

4.4.11 Template Manager

Introduction

The Template Manager module provides a set of template tags which are used to extend HTML templates with different functionality for generating dynamic HTML content. Template Manager is responsible for parsing these HTML templates for Pulsar tag. The set of tags and the tag functionality is provided also by Template Manager.

Pulsar Tags

<pulsar:echo ...>

The echo-tag is used to output variables from the Scope.

Attributes for the echo-tag:

Attribute	Description
name	The name of the variable in the scope that will be printed
urlencode	Set this attribute to `true` if the output should be url-encoded (percent-encoded)

<pulsar:if ...>

The if-tag is used to make output based on a condition.

Attributes for the if-tag:

Attribute	Description
test	A boolean expression that will be tested
eval	A boolean expression what will be evaluated with Bean Shell
true	This content of this attribute will be the result of the tag if the expressions is true
false	This content of this attribute will be the result of the tag if the expressions is false
file	If the expression is true, the template specified will be included and parsed for other pulsar tags, as it was included by the pulsar:include-tag

The test and the eval attribute can't be used together. One of them can only be used in every pulsar:if-tag.

Note that if the expression evaluates to true everything between the enclosing tag will be outputted. The content will be parsed and can contains other Pulsar-tags.

```
<pulsar:if test="flag">
    This is outputted if flag is true
    <pulsar:method service="service" method="
thisMethodIsCalledIfFlagIsTrue"/>
</pulsar:if>
```

<pulsar:method ...>

The method-tag is used to execute a Java-method annotated with `@PulsarMethod`.

Attributes for the method-tag:

Attribute	Mandatory	Default	Description
module	No	The same module as the template is loaded from.	The name of the module where method exists.
service	Yes		The name of the service where the method exists.
method	Yes		The name of the method which is called.
stylesheet	No		A name of a stylesheet which the XML from the method will be transformed with

Parameters to the method

Parameters to the method is coming from the request scope. If parameters are to be overridden , values can be passed to the methods argument by using the `<param name="..." value/variable="..."/>`-tag, as the example below.

```
<pulsar:method service="service" method="method">
    <param name="param1" value="value1"/>
    <param name="param2" variable="variable2"/>
</pulsar:method>
```

Parameters to the stylesheet

Parameters to the stylesheet is coming from the request scope the same way as the parameters to the method. If parameters are to be overridden, this can be done by using the <param name="..." value/variable="..."/>-tag, but a reference to the stylesheet from a separate tag must be declared.

```
<pulsar:method service="service" method="method">
    <stylesheet name="stylesheet.xsl>
        <param name="param1" value="value1"/>
        <param name="param2" variable="variable2"/>
    </stylesheet>
</pulsar:method>
```

Different parameters to the method and the stylesheet

Different parameters can be specified to the method and the stylesheet.

```
<pulsar:method service="service" method="method">
    <param name="param" value"This value goes only to the method"/
>
    <stylesheet name="stylesheet.xsl>
        <param name="param" value="This value goes only to the
method"/>
 </stylesheet>
</pulsar:method>
```

<pulsar:stylesheet ...>

The stylesheet-tag is used to execute a XSL-stylesheet without to call any java method. The XML-provided to the XSLT is:

```
<dummy/>
```

The provided XSL-stylesheet can use variables from the scope as input parameters.

Attributes for the stylesheet-tag:

Attribute	Mandatory	Description
name	No	A name of a stylesheet which the XML from the method will be transformed with

<pulsar:include ...>

The include-tag is use to include a separate template. The included file is parsed for pulsar-tags .

Attributes for the include-tag:

Attribute	Mandatory	Description
file	No	Name of the template to be included.
module	No	The name of the module where template exists.
template	No	The name of the template to be included

NOTE: Either use the file-attribute OR the module-attribute together with the template-attribute.

<pulsar:permission ...>

The permission-tag is use to do conditionally output, based on the current users permissions.

Attributes for the permission-tag:

Attribute	Mandatory	Default	Description
module	No	The same module as the template is loaded from.	The name of the module where the permission should be tested in.
type	Yes		The path of the resource as specified in resources.xml.
path	Yes		The type of the resource as specified in resources.xml.

Example:

```
<pulsar:permission type="template path="page1.html">
This is outputed if the user has permission to page1.html. The
section can contain other pulsar-tags that are parsed.
</pulsar:permission>
```

4.4.12 User Manager

Introduction

User Manager provides interfaces with the functionality to manage users and groups in Pulsar. User Manager can create, search or retrieve attributes for specified users and groups.

Relations between user to groups and groups to groups is also managed by User Manager. The relation structure is hierarchical, starting with a root group or user.

The actual backend for user and group data is pluggable through the `UserPlugin` interface. Multiple plugins may be used in one Pulsar instance.

Features

Plugin support

User Manager provides a unified view of users and groups from different sources. Plugins can be developed to add new sources of users and groups to the system. Each plugin specified it's own capabilities, for example if it supports updates or is read only. Usually extending `AbstractUserPlugin` is a good starting point when developing a new user plugin.

Budled plugin: Pulsar Default Plugin

The Pulsar Default Plugin is always available ant has the following groups defined:

Group	Description
everyone	Contains all users.
identified	Contains all logged in users.
anonymous	Contains all anonymous (not logged in) users.
none	Contains no users.

Bundled plugin: User XML Plugin

The XML plugin provides a simple XML format for providing using and groups data. The plugin also supports updating of the information.

Cascading (hierarchical) dependencies

If User/Group A is member of group B, and group B is member of group C then A is a member of C.

Attributes

Each user or group object has a set of mandatory attributes but may also have extra attributes as specified by a specific plugin.

Configuration

General notes on configuration.

4.5 Migrating from earlier versions

4.5.1 Introduction

This section describes the most important things that needs to be considered when migrating from Pulsar 2 to Pulsar 3.

4.5.2 Multiple services per module

In Pulsar 2 each module/service could only publish one service interface. In Pulsar 3 each module may publish any number of services defined by the same or different interface classes. When migrating from Pulsar 2 the old service interface will have to be extracted into a separate Java interface and published specifically. Since the service now needs to be specified separately from the module this affects calls to Pulsar methods both in templates and direct method calls, through HTTP POST/GET and javascript methods.

What was called service in Pulsar 2 now has to be referenced through both a module and a service. Both modules are uniquely identified through their symbolic name and a service instance is identified through the module symbolic name together with the symbolic name of the service (which is the same as the fully qualified Java interface name).

4.5.3 Dependency injection

Pulsar 3 used dependency injection with Guice. Dependency injection is a very important part of Pulsar since it is the primary channel for modules to exchange information and functionality, through services. Dependency injection can be done in different manners but the most usual way to do it is to consume and inject a service.

When migrating from Pulsar 2 the old service interface classes is replaced with service injection in the constructor.

The dependency injection model and dynamic modularity generally clashes with using static class member variables and static methods which initialized these variables. To utilize dynamic modularity and dependency injection static member variables needs to be converted into singleton classes which are then injected into the objects that needs them. This conversion not only allows for dependency injection and dynamic modularity but also makes the code testable.

4.5.4 Module activator

Pulsar 2 used the service class to interact with the lifecycle model. In Pulsar 3 the module lifecycle is implemented in a special activator class which needs to extend `PulsarActivator`.

4.5.5 Semantic versioning

To develop and release in semantic context semantic versioning must be followed in order for the application to behave properly. Semantic versioning is a key to create well defined dependencies in when developing in pulsar. See Concepts.

4.5.6 Strictly defined API classes

All classes (interfaces and value objects) must be places in an "api"-package in Pulsar 3. Only classes in the "api"-package will be available to other modules. This means that services which should be available as Pulsar methods has to be places in "api"-packages and published as services.

4.5.7 Use SLF4J for logging

Pulsar 3 uses SLF4J instead of standard java logging (JUL). Best practice is to provide every class with its own SLF4J logger. See Logging for developers.

4.5.8 Changes to core module APIs

Database Access changes

The Database Access module has been updated and is no longer backwards compatible with Pulsar 2. The service used is now `DatabaseAccessPool` and the usage of a poolName parameter is no longer supported.

4.5.9 Coding details

pulsar:method - calls från XSL

In Pulsar2 you made a pulsar:method call with **two** arguments: `pulsar:method('method', true/false)` where the second parameter indicated if the XSL should be cachable. In Pulsar 3 you only need **one argument:** `pulsar:method('method')`. When pulsar methods are called from the XSL the result of the XSL transformation will never be cached.

Modules and services

Since Pulsar 3 redefines services and modules any reference to a service in Pulsar 2 needs to be expressed as a reference to a service for a specific module in Pulsar 3. To make the migration easier Pulsar 3 allows each module to define a default service, which means that only the module and method is needed to invoke a default service.

4.5.10 Migrating published Web Services

When migrating existing Web Services from Pulsar 2 to Pulsar 3.x, with backwards compatibility , the following concerns needs to be adressed:

1. Pulsar 2 used AEGIS data binding per default, Pulsar 3 uses JAXB per default.
2. The default endpoint URL is changed in Pulsar 3, to avoid service name clashes. In Pulsar 2 the endpoint was "http://host/pulsar/ws/WSExample", the default endpoint created in Pulsar 3 would be "http://host/pulsar/ws/se.dse.pulsar.examples.wsexample.WSExample".
3. The target namespace of the generated WSDL may change if the package structure is changed. Any change of target namespace will break backwards compatibility.
4. The service name may affect backwards compatibility, this can be adjusted in the same way as the namespace.
5. Method parameters were defines as NOT NILLABLE by default in Pulsar 2. Pulsar 3.0 allows for all parameters to be NULL by default.

The following code example shows how to customize your service publication and service definition to be backwards compatible with Pulsar 2.0:

```
// publish the service with AEGIS data binding in your module
activator at the location http://host/pulsar/ws/WSExample
 i_contextBuilder.publish(WSExample.class)
                 .usingClass(WSExampleImpl.class)
                 .withConfigurator(lookupService(
WSConfiguratorFactory.class).configure()
                     .databinding(
WSConfiguratorFactory.DataBindingType.AEGIS)
                     .serviceAddress("WSExample"));
```

To specify the target namespace the standard JAXWS annotations are used on the interface:

WSExample.java

```
@WebService(targetNamespace = "http://my.custom.namespace",
serviceName = "WSExample")
public interface WSExample {
    @WebMethod
    String hello(@WebParam(name = "name") String i_name);
}
```

To further customize the AEGIS data binding (field mapping etc.) and controlling the NILLABLE schema attribute aegis.xml files can be used. For example to make the parameter to the hello method above not NILLABLE the following file is created next to the Java interface source.

WSExample.aegis.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings>
    <mapping name="WSExample">
        <method name="hello">
            <parameter index="0" nillable="false"/>
        </method>
    </mapping>
</mappings>
```

4.5.11 Migrating files in the data area

In Pulsar 2 each module could use a data area in the `pulsar-dir/service-dir/data` directory to read and write module specific data, usually application specific configuration files. In Pulsar 3 the same functionality exists but the data area is located at `pulsar-dir/data/module-symbolic-name` instead.

5 Development guides

5.1 Planning a module system

5.1.1 Naming a module

Advice on how to structure the modules into directories and name them (symbolic name, package names and directory names)

5.1.2 Dependencies should always be one-way

Give any two modules in the system they should either have no dependency or one directed dependency, direct or transient. It is possible to create circular dependencies and direct two-way dependencies between modules but this is highly discouraged because of the added complexity it brings (during development, continuous integration and deployment). If two-way or circular dependencies appear in your design you should refactor the involved modules so that the shared functionality is moved into a new or existing third, separate, module on which both the original modules can depend.

Given that the above practice is adhered to your modules dependencies can be described as a directed graph without any loops. This simplifies reasoning about the system of modules which is a huge benefit during system evolution.

5.1.3 Splitting an application into sub modules

When converting a monolithic application into modules the best practice is to first move the entire application into a single module and verify that it works as before. The next step is to select one suitable part and refactor that into a module. The extraction can be divided into these steps:

- Identify a block of functionality which is cohesive enough to be able to be encapsulated in a separate module. Also consider deployment requirements. Functionality that is not used by all deployments of a systen might be a good candidate since it can then only be deployed when it is actually needed.
- Create the new module. (see Creating a new module)
- Design the API of the new module. What methods of interaction should the module provide. Possible methods include Java API or exposure of resources, such as common HTML templates, XSL stylesheets or XML resources.
- Migrate the code and resources into the new module directory.
- Extract the API from the private code.

- Consume the new API where needed in the main application.

5.2 Creating a new module

5.2.1 Introduction

This section describes the fundamental steps that is involved when creating a new module in Pulsar.

- Setting up a Pulsar development environment
- Creating the required folder structure
- Building a module artifact
- Testing a module
- Deploying a module

5.2.2 Steps for creating a new module

The steps describes below are listed in a chronological manner where the first topic is a wise choice to start with.

Setting up the Pulsar development environment

In Pulsar 3.0 the development environment will be available as a preconfigured workspace for InterlliJ IDEA. In future versions Pulsar will include a Developer Tools module which will be able to create development projects for the most popular IDEs or a stand alone setup without IDE configuration.

Required module folder structure

Create a proper folder structure as follows:

- module-directory/
  - conf
  - lib/
    - exported (optional)
    - internal (optional)
    - native (optional)
  - resources
    - static

    - stylesheets
    - templates
  - script
  - src
  - src-test

This structure is required for the Pulsar tools to work. For more information on the directory structure see Development Reference / Directory Structures.

Writing module code

The Java code is placed in the `src` directory. Make sure that each module uses a unique package name. Having several modules use the same package will result in dependency conflicts and is highly discouraged.

Using other modules

When importing APIs from other modules you need to have these modules available both during the phases of compilation, test, packaging (to establish version dependencies) and in runtime. Pulsar 3.0 currently provides dependencies using the build files. To specify module dependencies use the `dependencies.compile` attribute in the build file, see example below.

Writing tests

Tests should be seen as a natural part of the development process. A good manner is to follow a test-driven development where tests are defined according to contract before starting to write the code. The Pulsar scripts assume that JUnit will be used for unit testning.

In Pulsar 3.0 only plain unit tests are supported, however in Pulsar 3.1 a framework for integration testing will be available. Using the framework it will be easy to create integration tests which involves several modules interacting with each other.

Test classes should be places in the src-test folder.

Creating a build script

To be deployable into a Pulsar framework the module needs to be packaged. Packaging is done with an Ant build script. The script has to be created in the `<MODULEDIR>/script` directory and called `build.xml`.

```
<project name="my_module" default="dist" basedir="..">
    <import file="../../../../script/build.macro.xml"/>
    <pulsar-module
```

```
            dir="."
            pulsar.dir="../../../"
            dependencies.compile="my_alfa_module, my_bravo_module"
            />
</project>
```

Running tests

The testing of a module is done by running the ant task `module.test` which is provided by `build.macro.xml`.

Building a module artifact

The module artifact (jar file) is created by running the ant task `module.dist` which is provided by `build.macro.xml`. By default javadoc is automatically generated for the API part of the module, this behavior may be overridden using the attribute `generate-javadoc` attribut of the `pulsar-module` tag in the build file.

Deploying a module

In a development environment deployment is usually handled implicitly. Pulsar has a development mode which provides automatic upgrades. This means that when Pulsar detects a newer module version in any of the current repositories it will automatically update that module, and all required dependencies. The module will be stopped, updated and restarted automatically, together with all the modules that depend on it.

In a remote deployment of Pulsar a module is deployed by adding the module artifact it to a HTTP repository or by packaging the module (usually together with other modules) into a repository package file (containing both modules and the repository index file). The repository package is then transfered to the remote Pulsar deployment and placed in at the root level of the installation. The module then is automatically available for installation or upgrade in the Admin Console / Install utility.

5.3 Logging for developers

5.3.1 Introduction

The choice of logging framework is not dictated by Pulsar. Pulsar is configured to support logging with the most common logging frameworks, Java Util Logging (JUL), Java Commons Logging (JCL) and Log4J. However, to get the most out of the logging functionality it is recommended to use the framework SLF4J (Simple Logging for Java). Internally all logging through the supported frameworks is automatically routed through SLF4J which is then used as the main framework though which log output is produced and routed to different targets.

5.3.2 Implementing logging

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Clazz {
    private final static Logger logger = LoggerFactory.getLogger(
Clazz.class);

    void method() {
        logger.warn("[method] Something is not right!");
        logger.info("[method] Information about what's happening")
;
        logger.debug("[method] Debug info");
        logger.trace("[method] Detailed information");
        try {
            ...
        }
        catch (Exception e) {
            logger.warn("[method] Exception ...", e);
        }
    }
}
```

Migrating from JUL to SLF4J

In Pulsar 2 java.util.logging (JUL) was the standard logging framework. However, the migration is quite straightforward and the following best practice can be used when translating logger method calls.

Old Logger (java.util.logging.Logger)	New Logger (org.slf4j.Logger)
severe(...)	error(...)
warning(...)	warn(...)
info(...)	info(...)
config(...)	debug(...)
fine(...)	debug(...)
finer(...)	trace(...)

Old Logger (java.util.logging.Logger)	New Logger (org.slf4j.Logger)
finest(...)	trace(...)
log(Level,...)	No dynamic selection of logging is available in SLF4J. This is probably by design since it is not good practice to select log level in runtime. Logging filters should be configured using log configuration instead. When migrating a log level needs to be selected for each instance.

Note that the SLF4J logger only has five different levels.

5.3.3 Logging features

Changing default log level

The default log level is specified in the `level` attribute in the `root`-tag in the `logback.xml` file . Setting the default level to `DEBUG` or `TRACE` is not recommended since massive amounts of logging information will be generated. Instead set the logging levels for specific packages of interest, as described below.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration ...>
    ...
    <root level="INFO">
        <appender-ref ref="STDOUT" />
        <appender-ref ref="FILE" />
        <appender-ref ref="FILE_WARNINGS" />
    </root>
    ...
</configuration>
```

Setting the logging levels for specific packages

You can set the log level for a module by setting it's corresponding package logging level in `logback.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration scan="true" scanPeriod="10 seconds" debug="false">
    ...
    <!-- Setting levels for different packages -->
    <logger name="se.dse.pulsar.module.permissionmanager" level="
DEBUG"/>
    <logger name="se.dse.pulsar.module.sessionmanager" level="
TRACE"/>
```

```
    ...
</configuration>
```

Splitting logging into different files

By default the log is split up into two two different type off log files;

- `pulsarN.log` (N=0..9) - This i a rotating log where pulsar0.log is the newest and pulsar9.log is the oldest log file.
- `pulsar-warnings0.log` - This log files contains only warnings and error.

Tead more about how to split the log files here.

Automatic reload of configuration file

By default, the configuration file will be scanned for changes once every 10 seconds, so there is no need to restart Pulsar if you change the logging config.

5.3.4 Logging configuration

The logging framework is configured through the file `<PULSARDIR>/conf/logback.xml.`

Example of `logback.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration scan="true" scanPeriod="10 seconds" debug="false">
    <appender name="STDOUT" class="
ch.qos.logback.core.ConsoleAppender">
        <withJansi>false</withJansi>
        <encoder class="
ch.qos.logback.classic.encoder.PatternLayoutEncoder">
            <pattern>%d{HH:mm:ss} [%thread] %-5level %logger{36} -
%msg%n</pattern>
        </encoder>
    </appender>
    <appender name="FILE" class="
ch.qos.logback.core.rolling.RollingFileAppender">
        <append>true</append>
        <file>logs/pulsar0.log</file>
        <rollingPolicy class="
ch.qos.logback.core.rolling.FixedWindowRollingPolicy">
            <fileNamePattern>logs/pulsar%i.log</fileNamePattern>
            <minIndex>1</minIndex>
            <maxIndex>9</maxIndex>
        </rollingPolicy>
        <triggeringPolicy class="
ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">
```

```
            <maxFileSize>1MB</maxFileSize>
        </triggeringPolicy>
        <encoder>
            <pattern>%date{yyyy-MM-dd HH:mm:ss} [%thread] %-5level
%logger{35} - %msg%n</pattern>
        </encoder>
    </appender>
    <appender name="FILE_WARNINGS" class="
ch.qos.logback.core.rolling.RollingFileAppender">
        <append>true</append>
        <filter class="
ch.qos.logback.classic.filter.ThresholdFilter">
            <level>WARN</level>
        </filter>
        <file>logs/pulsar-warnings0.log</file>
        <rollingPolicy class="
ch.qos.logback.core.rolling.FixedWindowRollingPolicy">
            <fileNamePattern>logs/pulsar%i.log</fileNamePattern>
            <minIndex>1</minIndex>
            <maxIndex>1</maxIndex>
        </rollingPolicy>
        <triggeringPolicy class="
ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">
            <maxFileSize>1MB</maxFileSize>
        </triggeringPolicy>
        <encoder>
            <pattern>%date{yyyy-MM-dd HH:mm:ss} [%thread] %-5level
%logger{35} - %msg%n</pattern>
        </encoder>
    </appender>
    <!-- Make some talkative loggers quiet during normal startup -
->
    <logger name="org.eclipse.jetty" level="WARN"/>
    <logger name="org.apache.cxf" level="WARN"/>
    <logger name="org.apache.aries" level="WARN"/>
    <logger name="org.ops4j.pax" level="WARN"/>
    <logger name="org.apache.felix.fileinstall" level="WARN"/>
    <!-- Setting levels for different packages -->
    <logger name="se.dse.pulsar.module.permissionmanager" level="
DEBUG"/>
    <logger name="se.dse.pulsar.module.sessionmanager" level="
TRACE"/>
    <root level="INFO">
        <appender-ref ref="STDOUT" />
        <appender-ref ref="FILE" />
        <appender-ref ref="FILE_WARNINGS" />
    </root>
</configuration>
```

5.3.5 References

1. Logback configuration - http://logback.qos.ch/manual/configuration.html

5.4 Working with Web Services and REST Services

5.4.1 Introduction

Pulsar provides the ability to publish both Web Services and REST services though the same service mechanism which is used to publish and consume Pulsar service. A service in Pulsar is defined by its Java interface and this is true also for Web Services and REST services. A Web Service or a REST service is a Pulsar service with extra annotations describing the service and some additional configuration which is supplied during the publishing of the service. This means that a service can be simultaneously published as a Pulsar service, Web Service and REST service.

5.4.2 Publishing a Web Service

Create the service interface in an API package and annotate it according to the JAX-WS standard, using the annotations specified in the `javax.ws` package.

```
package se.dse.pulsar.examples.wsservice.api;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
@WebService
public interface WSExample {
    @WebMethod
    String hello(@WebParam(name = "name") String i_name);
}
```

Create the implementation (`WSExampleImpl` according to the code example) in an internal package and publish it in the module activator `init()` method. Note, the implementation doesn't require any JAX-WS annotations.

```
package se.dse.pulsar.examples.wsservice;
import se.dse.pulsar.core.api.ModuleContextBuilder;
import se.dse.pulsar.core.api.PulsarActivator;
import se.dse.pulsar.core.api.configurators.RSConfigurator;
import se.dse.pulsar.examples.wsservice.api.WSExample;
public class WSExampleModuleActivator extends PulsarActivator {
```

```
    @Override
    public void init(ModuleContextBuilder i_contextBuilder) {
        i_contextBuilder
                .publish(WSExample.class)
                .usingClass(WSExampleImpl.class)
                .withConfigurator(lookupService(
WSConfiguratorFactory.class).configure());
    }
}
```

The Web Service will now be available as soon as the module has been installed and started. The service is automatically shut down when the module is stopped.

Web Services and their WSDL specifications are available at their respective endpoint addresses. In the example above the WSDL would be available at `http://localhost:8080/pulsar/ws/se.dse.pulsar.examples.wsservice.api.WSExample?.wsdl`.

Note, the WSConfigratorFactory uses the builder pattern to provide additional configuration options for the web service, such as XML binding style, custom addresses and more. The JAX-WS standard also allows for customization of the service namespace, service and endpoint names using annotations on the interface.

5.4.3 Publishing a REST service

Create the service interface in an API package and annotate it according to the JAX-RS standard, using the annotations specified in the `javax.ws.rs` package.

```
package se.dse.pulsar.examples.rsservice.api;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
@Path("example")
public interface RSExample {
    @GET
    @Path("hello/{name}")
    public String hello(@PathParam("name") String name);
}
```

Create the implementation (`RSExampleImpl` according to the code example) in an internal package and publish it in the module activator `init()` method. Note, the implementation doesn't require any JAX-RS annotations.

```
package se.dse.pulsar.examples.rsservice;
import se.dse.pulsar.core.api.ModuleContextBuilder;
import se.dse.pulsar.core.api.PulsarActivator;
```

```
import se.dse.pulsar.core.api.configurators.RSConfigurator;
import se.dse.pulsar.examples.wsservice.api.RSExample;
public class RSExampleModuleActivator extends PulsarActivator {
    @Override
    public void init(ModuleContextBuilder i_contextBuilder) {
        i_contextBuilder
                .publish(RSExample.class)
                .usingClass(RSExampleImpl.class)
                .withConfigurator(lookupService(
RSConfiguratorFactory.class).configure());
    }
}
```

The REST Service will now be available as soon as the module has been installed and started. The REST service may now be invoked through a browser using the URL `http://localhost:8080/pulsar/rs/se.dse.pulsar.examples.wsrsservice.api.RSExample/example/hello/merlin` (passing the parameter 'merlin' to the hello method). The service is automatically shut down when the module is stopped.

REST Services and their WADL specifications are available at their respective endpoint addresses. In the example above the WADL would be available at `http://localhost:8080/pulsar/rs/se.dse.pulsar.examples.rsservice.api.RSExample?_WADL`.

Note, the RSConfigratorFactory uses the builder pattern to provide additional configuration options for the web service, such as XML binding style, custom addresses and more. The JAX-WS standard also allows for customization of the service namespace, service and endpoint names using annotations on the interface.

5.5 Development troubleshooting

5.5.1 Introduction

The main source of information about the state of the running Pulsar framework are the log files. All the code which makes up Pulsar and it's core modules aims at producing relevant and succinct logging information to be used by developers or operations. Warnings and errors in the log should be treated seriously and during normal operations no warnings or errors should be present.

5.5.2 Increase the logging output to troubleshoot

Check the log for warnings. Under normal circumstances the log should not contain any warnings och errors. Read the message thoroughly and examine the call stack if available.

To be able to get more information you can change the log level for a module. This is specified in the config file `conf/logback.xml`. As a general rule start with activating the `DEBUG` level of logging and advance to `TRACE` if the debug level is not providing enough detail.

Example for setting Session Manager to log debug information.

logback.xml

```
<configuration ...>
    ...
    <logger name="se.dse.pulsar.module.sessionmanager" level="
DEBUG"/>
</configuration
```

For more information about logging see Development Reference / Logging for developers.

5.5.3 Use the debugger and set breakpoints

To be able to examine application flow and get information about variable, it is very handy to use the debugger in the IDE.

5.5.4 Using a command line shell to introspect

The Apache Felix project provides a runtime command shell, called GoGo which may be used to introspect Pulsar modules and their dependencies in runtime. To install and activate the GoGo shell on a Pulsar instance use the command line options below when starting Pulsar. The shell commands are documented by the Apache Felix project on their web site.

```
java -Dse.dse.pulsar.module.boot.forceprofiledeploy=true -jar
pulsar.launcher-v.v.v.jar default shell
```

5.5.5 Using Hot Swap to deploy minor code changes

If your IDE supports hot swapping classes in the running Java runtime this can be used as an efficient tool for correcting and retesting code without having to re-dist your modules using the build scripts.

5.5.6 Coding considerations

Using the correct annotations

Annotation problems can be very difficult to debug since the wrong type of annotation produce the same result as no annotation. Further if an annotation class is unavailable at runtime this also has the same effect as if no annotation was present, no error or warning is shown.

The `javax.inject` annotations `@Inject`, `@Named` should always be used instead of the corresponding `com.google.inject` alternatives. A good idea is to filter away the import suggestion the `com.google.inject` package.

Only interfaces may be published as services

When publishing services a Java interface must be used to represent the service. An implementation class may not be published directly.

6 Operations reference

6.1 Installation and upgrades

6.1.1 Installing a new Pulsar instance

To install Pulsar you need four files and JDK/JRE 7+ installed:

1. pulsar.launcher-v.v.v.jar
2. pulsar-modules-3.v.v.pulsar
3. pulsar-apps-3.v.v.pulsar
4. pulsar-endorsed-3.v.v.pulsar

Create the directory where you want the pulsar instance to be installed and copy the installation files into that directory.

Install and start Pulsar by invoking the launcher:

```
java -jar pulsar.launcher-v.v.v.jar
```

This will extract the Pulsar runtime into the subdirectory `.pulsar-runtime` and start the application server. Once the Pulsar modules are loaded a browser (using your default browser) window will automatically be launched and logged into the Admin Console application.

Creation of a Launcher link

When the Pulsar Launcher is started it always creates a link file `pulsar.launcher-link.jar`. This file is relinked whenever a different Launcher is started so that it always points to the Launcher which was last used. The purpose of the Launcher link is to provide a consistent filename over upgrades, which may be used to start Pulsar as a service or from a script. Without having to update the service definition or the script when the Launcher is upgraded or downgraded.

The link created is a hard link, this means that it is a proper file in all respects, but which points to the same inode as the linked file. (See you file system documentation for further information.)

Administrative (super user) password

The password for the super user, which is used to access Admin Console, is randomly generated as a part of the install. The password for the pulsar.superuser ('admin') can be changed directly after an installation through the Admin Console / Configuration screen for the pulsar.usermanager module. If the password is not changed it can be retrieved from the file `logs/admin-pwd.log`.

The default user name for the super user is 'admin' this can also be changed through the configuration of the pulsar.usermanager module.

6.1.2 Starting an existing Pulsar instance

To start an existing Pulsar instance from the command line, use the same command as used when installing. The Pulsar Launcher will detect the existing instance and launch it instead of reinstalling.

```
java -jar pulsar.launcher-v.v.v.jar
```

If Pulsar is installed as a system service the service control panel in the operating system may be used to start Pulsar (se Running Pulsar as a system service)

6.1.3 Installing, upgrading and uninstalling modules

Modules in Pulsar may be installed, upgraded or installed without restarting the Pulsar instance (Java process). The installation, upgrade and uninstallation is handled through a web interface of Admin Console / Install. The web interface also shows which module versions are currently installed and if any upgrades are available in the known repositories. See Operations reference / Repositories for more information on how Pulsar detects and handles module repositories (local and remote).

6.1.4 Running Pulsar as a system service

Note: To apply the autostart configuration Pulsar needs administrative privileges. On Windows this can be achieved by starting Pulsar in a Command Windows which has been started with the "Run as Administrator..." option.

Installation of Pulsar as a system service is handled by the pulsar.autostart module. The configuration of the service is done in the Admin Console / Config / pulsar.autostart. The service is installed when the autostart parameter is enabled. The service name default is "Pulsar" but this may be changed in the configuration.

Service Logging

The Windows service is installed using Apache Commons Daemon. Logging from the Common Daemon is done to a log file called commons-daemon.*YYYY-MM-DD*.log.

Starting the service from the command line

For troubleshooting purposes it might be necessary to start the service from the command line.

```
prunsrv run ServiceName
```

6.1.5 Upgrading the Pulsar Launcher

Modules in a Pulsar instance can be upgraded in runtime using the Admin Console, as described above. Upgrading the Pulsar Launcher on the other hand, requires the instance (the Java process) to be shut down and restarted. The Pulsar Launcher contains minimal functionality which is used to boot the system, load core modules and boot the rest of the system. Therefore it is expected that module upgrades will be far more frequent than Launcher upgrades, which should only happen very seldom. For this reason the Launcher is also versioned separately from the other Pulsar packages.

1) Before upgrading the Pulsar Launcher, make sure the Pulsar instance is not running (in a terminal or as a service).

2) Upgrade the Pulsar Launcher by copying the new version of the Launcher (`pulsar.launcher-v.v.v.jar`) to the Pulsar instance directory.

3) Update the Launcher link using the following command:

```
java -jar pulsar.launcher-v.v.v.jar --relink
```

4) Remove the old Launcher to avoid it from being used inadvertently after the upgrade.

Checking which version the Launcher link points to

The easiest way to check which version the Launcher link points to is to use the following command:

```
java -jar pulsar.launcher-link.jar --version
```

Reverting to an older launcher

If a Launcher needs to be downgraded the same procedure as when upgrading may be used.

6.1.6 Removing a Pulsar instance

To remove a Pulsar instance all that is needed is to delete the directory where Pulsar is installed. However you need to check the following before doing that:

- If you installed Pulsar as a Windows Service and want to remove that as well, first deactivate autostart in the Admin Console / Config / pulsar.autostart.
- If you have module configurations, or other instance persisted information you need to preserve, export this first. Module configurations can be exported in Admin Console / Config.
- Make sure Pulsar is not running.
- Delete the directory (including the sub-directory `.pulsar-runtime`

In the event of orphaned services in Windows, these may be cleaned up manually using the `sc` command, which is provided by Microsoft.

6.2 Repositories

6.2.1 What is a module repository?

A module repository is a set of Pulsar module artifacts (jar files created in a development environment or on a build server) together with an index (XML file) which describes the modules . The index contains extracted information from the artifacts, such as, module names, versions, exported packages, required packages, timestamps and paths to the module artifacts files. A module index can be built entirely from the module artifacts. Pulsar contains tools for creating these indexes.

6.2.2 Why use a module repository?

A module repository can be used to store several versions of the same module. Usually modules are never removed from the repository. New module versions are continually added as they are released. This is a very important concept when working with module dependencies and semantic versioning. Since one module may be compatible with certain versions of another module the Pulsar installer, which uses the repository, may select the latest compatible version instead of failing the install because of a broken dependency.

The repository concept together with the semantic versioning of modules provides a dynamic release system where modules are free to evolve independently, while at the same time module compatibility and configuration management is ensured in runtime.

6.2.3 Different forms of repositories

A repository may exist in different forms:

1. In a development environment the modules under development are treated as a repository. The index file is generated at `dist/repository.xml` and it will point to all the module artifacts that has been built (using `module.dist`) on the local machine.
2. When a repository is published over HTTP the same structure is used and the artifact files and the `repository.xml` is made available over HTTP from any web server.
3. When installations are made without an online repository a packaged form of the repository may be used. Here the artifacts and the index is packaged into a single file with the suffix `.pulsar`. Pulsar uses this way of distribution for it's own modules and endorsed libraries.

6.2.4 Adding repositories to a Pulsar instance

Pulsar repositories are added to an instance as .pulsar files or .pulsarlink files.

To connect a Pulsar instance to a pulsar repository file (.pulsar file) just copy the repository file to the root directory where Pulsar is installed/running. Pulsar automatically detects when repositories are added or removed. If a repository is temporarily unavailable, for example caused by a network outage, this will be logged to the pulsar.log. To see which modules are available for installation or upgrade through all currently available repositories, use the Admin Console, Install page.

To connect a Pulsar instance to a HTTP repository create a pulsar repository link (.pulsarlink file). The file is a plain text file, only containing a URL pointing to the repository index file (usually repository.xml). Example of a .pulsarlink file:

```
http://repo.example.org:8080/products/alfa/develop-repo/
repository.xml
```

6.2.5 Setting up a HTTP repository

To publish a repository over HTTP the following is required:

1. Any HTTP-server capable of serving repository content.

2. A directory structure with the modules to be published in the repository (.jar files).
3. An index file (repository.xml) generated by a module indexing tool (*scheduled to be bundled with Pulsar 3.x*).

On the instances where the repository should be consumed a .pulsarlink file, which points to the repository, needs to be added (see example above).

6.3 Configuration

6.3.1 Deployment profiles

Pulsar uses deployment configuration profiles to define sets of modules which can be installed by applying the profile.

The following profiles are bundled with Pulsar:

Profile name	Description
default	The default profile which is a base profile required to start the Pulsar infrastructure services and their dependencies. If no profile is specified this profile will be automatically used.
developer	The developer profile can be used during development of Pulsar modules. This profile enables automatic upgrades.
shell	Activates a command line shell (Apache Felix GoGo) which may be used for debugging purposes.

Applying a profile

Profiles affect the modules which are installed during installation and basic framework configuration parameters. Profiles are applied using the Pulsar Launcher. For example to install Pulsar and apply the developer profile use the following command:

```
java -jar pulsar.launcher-V.V.V.jar default developer
```

If Pulsar is already installed the profile options will not cause any modules to be installed unless the parameter `-Dse.dse.pulsar.module.boot.forceprofiledeploy=true` is used. For example, to add the shell profile to an existing instance use the following command when starting:

```
java -Dse.dse.pulsar.module.boot.forceprofiledeploy=true -jar
pulsar.launcher-V.V.V.jar default shell
```

Not that when specifying additional profiles the `default` profile must always be specified to load the Pulsar infrastructure.

A profile can not be "uninstalled" through the command line but the modules concerned may be uninstalled manually through the Admin Console / Install page.

6.3.2 Module configurations

Each module have a configuration associated with it in Pulsar. The configuration functionality is handled by the Configuration Manager module (se Developer reference). Operations concerning module configuration is performed in Console Admin / Configure page. The web based configuration services allows configuration changes in runtime. Configurations may also be imported or exported as text files.

When a configuration change is saved in Console Admin / Configure the affected module will be automatically restarted by Pulsar.

Notes on configuration

When the text "NULL" is used as a configuration value the Java value `null` will be assigned to the corresponding configuration parameter in the Java code.

6.4 Requirements and performance

6.4.1 Performance monitoring

Pulsar provides a runtime monitoring application called Performance Viewer which presents web oriented performance statistics in the Admin Console. For monitoring of memory usage Pulsar provides the Memory Viewer app (since Pulsar Apps 3.0.2). Both the Performance Viewer and Memory Viewer are part of the Pulsar Apps package.

6.4.2 Hardware and software requirements

Pulsar 3.0 requires Java JDK/JRE 7 or above and may be run on any hardware on which the JDK can be used. Either 32-bit or 64-bit JDK/JRE can be used.

Pulsar 3.0 requires 200 MB of disk space for an initial installation.

Pulsar 3.0, including dependencies, has an initial memory footprint of around 100 MB (Java Heap Space). The minimal memory requirement for running the JDK/JRE can be found in the documentation of the JDK/JRE.

6.5 Logging

6.5.1 Logs created by Pulsar

By default Pulsar generates three types of logs in runtime:

- General Pulsar log - `pulsar-instance/logs/pulsarN.log` and the console
- Pulsar warnings log - `pulsar-instance/logs/pulsar-warnings0.log`
- HTTP log - standard HTTP logging to be used for traffic analysis `logs/http/YYYY_MM_DD.http.log`
- Service log (used when Pulsar is started as an operating system service) `logs/service/commons-daemon.YYYY-MM-DD.log`

6.5.2 General log

The general log contains all the logging information generated by Pulsar and all modules running in the system, including third party and user modules. The level of log information can be controlled by configuration and may be adjusted in runtime without restarting the Pulsar instance. By default the general log is rotated when it´s size exceeds 1MB, a maximum of ten log files are kept.

6.5.3 Pulsar warnings log

The warnings log only contains warnings or error messages. These are also available in the general log. The purpose of the warning log is to be able to check warnings and errors without having to scan the entire general log. During normal operations the warning log should be empty and any warnings or errors should be acted on. By default the warning log is rotated when it´s size exceeds 1MB, only the latest warning log file is kept.

6.5.4 Service log

The service log is only created when Pulsar is installed as an operating system service. This log can be used to troubleshoot service configuration errors such as permissions etc.

6.5.5 Configuration of logging

For details on how to configure logging in runtime see Developer Reference / Logging for developers.

6.6 Operations troubleshooting

6.6.1 Recovering from errors during installation or upgrade

When dependencies are unavailable, if a package file is damages or if a HTTP repository suddenly no longer is available because of a network failure, the installation or upgrade of Pulsar might not be able to complete successfully. In this case the instance might be in a partially installed state where the administrative tools are unavailable. The following approaches may be used to recover enough to get the administrative tools online again.

Restart and reapply the deployment profile

By reapplying the deployment profile Pulsar will try to install all the default modules again, the same way as when the instance was first installed. Reapplying the profiles is generally a safe operation. Modules will be upgraded if newer versions exists and no modules or persisted data (like configurations) are removed. For more information on deployment profiles see Configuration .

```
cd my-pulsar-instance
java -jar pulsar.launcher-v.v.v.jar -
Dse.dse.pulsar.module.boot.forceprofiledeploy=true
```

Reinstalling Pulsar

To completely reinstall an existing Pulsar instance, delete the `.pulsar-runtime` directory and then rerun the Pulsar Launcher. Note, this will remove any configuration and other information persisted by all modules in the instance.

Reinstalling Pulsar on Windows

```
cd my-pulsar-instance
rm -rf .pulsar-runtime
java -jar pulsar.launcher-v.v.v.jar
```

Reinstalling Pulsar on Linux/OSX

```
cd my-pulsar-instance
rmdir /s /q .pulsar-runtime
java -jar pulsar.launcher-v.v.v.jar
```

Advanced - Manual recovery using a command line shell

Modules may be installed and started using a command line shell which can be activated after installation. See Development troubleshooting / Using a command line shell to introspect for instruction on how to activate the shell.

6.6.2 Recovering from runtime errors

Restarting modules

Sometimes restarting a module may recover from a module local failure mode. Individual modules can be restarted in runtime, using the Console Admin / Install page.

Restarting Pulsar

Runtime errors may be caused by a huge number of different reasons. Some of the reasons may be memory leaks, network failures, software bugs etc. Sometimes a restart of Pulsar may cause modules to recover from their failure mode. Restarting Pulsar will ensure that all modules are restarted.

If Pulsar is installed as a system services the service console should be used to perform the restart. If the Pulsar process (java process) is terminated the running module will not shut down in a controlled fashion, which may cause further problems. Thus, terminating the process should only be used as a last resort.

Restarting Pulsar will not affect which modules are installed or their current configuration.

6.6.3 Operations errors

Bundles can't be properly uninstalled after update or uninstall

This problem may occur during Pulsar startup or when a module fails to uninstall.

Symptoms

The log contains one or both of the following messages:

- `ERROR: Could not purge bundle`
- `ERROR: Unable to re-install`

Cause:

The cause for this problem is that Pulsar is unable to find or modify internal files or directories from the `.pulsar-runtime` directory. Most often this is caused by files being locked or removed by another process.

Resolution or workaround:

First, make sure that multiple Pulsar instances are not running against the same installation directory. Then, also check that any anti virus or backup software is not locking the files.

7 Support

7.1 Collecting information for a support request

A support issue request must contain the following items:

1. Subject - a short caption describing the problem.
2. Description - problem description, when did the problem occur and how can it be reproduced.
3. Timestamp - when did the problem occur (used to cross reference any logs).
4. Versions used - which versions of Pulsar (launcher, apps, modules, endorsed) were used.
5. Environment - Windows, Linux or OSX, which JDK/JRE version.
6. Contact information - your contact information.

A support issue request should also contain the following items:

1. Log files (pulsar-warnings0.log, pulsar0.log)
2. Screen shots if the problem manifests in the user interface.

7.2 Contact

Customers with support entitlements may report issues and suggestions through our homepage at http://www.dse.se. Click the "Support" link in the main menu to reach our support form.

For general inquires contact info@dse.se or support@dse.se.

8 Licenses

8.1 Documentation License

8.2 Pulsar License

Licenses for any use of the Pulsar software, or any of it's parts, tools, scripts and documentation , is currently negotiated on a per customer basis. Please contact us for further information.

Pulsar is currently not licensed under a Open Source license, however this is something we are looking into and we hope to be able to provide an Open Source licensing option in the near future.

8.3 Open Source Licenses

Pulsar is powered by several Open Source tools and libraries (using their binary distributions).

We humbly acknowledge the great work produced by the committers on the following projects:

Libraries	Vendor / Organization	License
Apache Felix (and sub projects) Apache Ant Apache Lang Commons Apache IO Commons Apache Logging Commons Apache XML Commons Apache XML Graphics Commons Apache FOP Apache Batik Apache Avalon Apache Xalan Apache Xerces Apache CXF Apache DOSGi Apache Tika	Apache Software Foundation	http://www.apache.org/licenses/LICENSE-2.0

Libraries	Vendor / Organization	License
Apache PDFBox Apache POI Apache Commons Codec		
OSGi Compendium 4.3	OSGi Alliance	http://www.osgi.org/Main/OSGiSpecificationLicense
Google Guice	Google	http://www.apache.org/licenses/LICENSE-2.0
javax.inject	JCP	http://www.apache.org/licenses/LICENSE-2.0
Simple Logging Facade for Java (SLF4J)	QOS.ch	Copyright (c) 2004-2013 QOS.ch. All rights reserved.http://slf4j.org/license.html
LogBack	QOS.ch	EPL v1.0 and the LGPL 2.1, http://logback.qos.ch/license.html
Glassfish (javax.mail, javax.servlet)		https://glassfish.java.net/public/CDDL+GPL.html *(WITH "CLASSPATH EXCEPTION")*
aQute BND	aQute	http://www.apache.org/licenses/LICENSE-2.0
Oracle JDBC Driver	Oracle	http://www.oracle.com/technetwork/licenses /distribution-license-152002.html
Transloader	https://code.google.com/p/transloader/	http://www.apache.org/licenses/LICENSE-2.0
Bean Shell	http://www.beanshell.org/	http://www.beanshell.org/license.html
JavaMail(TM) API Design Specification	Oracle	https://glassfish.java.net/public/CDDL+GPL_1_1.html *(WITH " CLASSPATH EXCEPTION")*
JDOM	http://www.jdom.org/	Copyright (C) 2000-2004 Jason Hunter & Brett McLaughlin. All rights reserved. *(Apache-style open source license)*
DOM4J	MetaStuff, Ltd	BSD Style license http://dom4j.sourceforge.net/dom4j-1.6.1/license.html
Elastic Search	http://www.elasticsearch.org	http://www.apache.org/licenses/LICENSE-2.0

9 Release Notes

9.1 Release 3.1.0

9.1.1 Changes since 3.0.5

[PULSAR-631, PULSAR-638, PULSAR-639, PULSAR-640, PULSAR-641, PULSAR-645] - Added APIs for and implementations of search engine, scanners and indexers.
[PULSAR-697] - Deterministic definition of context class loader to bundle classloader in PulsarActivator life cycle methods
[PULSAR-698] - Monitoring the creation of new threads in all PulsarActivator life cycle methods
[PULSAR-701] - Added new bundle dependencies for search to endorsed package
[PULSAR-719] - Documentation of Search Module

Bug fixes

[PULSAR-614] - When ClientAccess configuration is updated through AdminConsole the Pulsar servlet becomes unavailable
[PULSAR-707] - InstallerUI can't show modal install dialog in IE 9

9.1.2 Known Issues

[PULSAR-713] - Logs are not always rotated when size limit is reached
[PULSAR-703] - Current Users in Admin Console show shows wrong module and action
[PULSAR-662] - Pulsar upgrades may cause an inconsistent dependency state
[PULSAR-654] - When deactivating autostart Pulsar is stopped which causes deadlock
[PULSAR-628] - StackOverflowError pulsar.module.dataaccess.CacheHandler.sqldivide
[PULSAR-625] - New methods in pulsar/js are not updated on reload, only on forced reload
[PULSAR-610] - Updated modules/repositories are not always automatically updated by automatic upgrade
[PULSAR-608] - Templates is affected by the default file.encoding, it should only be affected by config in Template Manager
[PULSAR-575] - OutOfMemoryError: PermGen space after updating a module (leb.leb-portal) around ten times in development mode
[PULSAR-507] - Caused by exception is hidden when a ClassDefNotFound exception is thrown in a library constructor.
[PULSAR-485] - Can't dist module within the same minute
[PULSAR-451] - When an invalid install-profile is specified the default should be used.
[PULSAR-417] - Analysis - Stylesheet is picked up from the wrong module in <pulsar:method module="..." stylesheet="..."/>

9.2 Release 3.0.5

9.2.1 Changes since 3.0.4

[PULSAR-668] - Changed location of HTTP-logs och commons-daemon log files to sub directories

Bug fixes

[PULSAR-624] - Module upgrades sometimes causes two versions of the module to be installed
[PULSAR-684] - Exception when instantiating a web service client, added test cases.

9.3 Release 3.0.4

9.3.1 Changes since 3.0.3

Bug fixes

[PULSAR-687] - MessageManager - Can't add attachment, exception javax.activation.UnsupportedDataTypeException.

9.4 Release 3.0.3

9.4.1 Changes since 3.0.2

[PULSAR-602] - Support for Launcher upgrades when autostart service is used

Bug fixes

[PULSAR-665] - Out of Memory when using iterable service provider
[PULSAR-679] - MemoryViewer - Exception thrown when trying to set threshold on unlimited pools.

9.5 Release 3.0.2

9.5.1 Changes since 3.0.1

[PULSAR-595] - Support for selecting SMTP port, username and password in MessageManager
[PULSAR-669] - Support for monitoring of memory usage

Bug fixes

[PULSAR-670] - MessageManager can't find javax.net.ssl, which causes send to fail

9.6 Release 3.0.1

9.6.1 Changes since 3.0.0

[PULSAR-249] - Write draft of Operations reference
[PULSAR-609] - Improved boot logging to make troubleshooting/support easier
[PULSAR-632] - Add JVM memory options to autostart (service)
[PULSAR-642] - Always log which ports are used and when they are updated
[PULSAR-623] - Release adjustments of documentation

Bug fixes

[PULSAR-613] - Can't rename ServiceName in Autostart
[PULSAR-633] - NPE in Admin Console / PerformanceViewer
[PULSAR-634] - Template cache grows