

# **Tigase Development Guide**

**Tigase Team**

---

# **Tigase Development Guide**

Tigase Team

---

# Table of Contents

1. Basic Information .....	1
Tigase Server Elements .....	1
Components .....	1
Plug-ins .....	2
Connector .....	2
Data, Stanzas, Packets - Data Flow and Processing .....	2
2. Hack Tigase XMPP Server in Eclipse .....	4
Requirements .....	4
Installation .....	4
Linux .....	4
Windows .....	4
Setup .....	10
API changes in the Tigase Server 5.x .....	15
3. Server Compilation .....	17
Tigase XMPP Server 5.2.0 and later - Compilation and Generating Distribution Packages .....	17
Distribution Packages .....	17
Building Server and Generating Packages .....	17
Running Server .....	18
Tigase Packages Dependency Change - Server Compilation Version 4.x or Later .....	18
4. Component Development .....	20
Component Implementation - Lesson 1 - Basics .....	20
Component Implementation - Lesson 2 - Configuration .....	23
Component Implementation - Lesson 3 - Multi-Threading .....	27
Component Implementation - Lesson 4 - Service Discovery .....	31
Component Implementation - Lesson 5 - Statistics .....	37
Component Implementation - Lesson 6 - Scripting Support .....	41
Component Implementation - Lesson 7 - Data Repository .....	47
ConfigRepository .....	47
RepositoryFactory .....	47
Component Implementation - Lesson 8 - Startup Time .....	47
Configuration API .....	48
Introduction .....	48
Component Startup Sequence .....	48
Configuration API .....	49
getDefaults() .....	49
setProperties() .....	50
Useful Presets .....	50
Global Configuration Settings .....	50
Packet Filtering in Components .....	51
The Packet Filter API .....	51
Configuration .....	52
EventBus API in Tigase .....	53
EventBus API .....	53
Distributed EventBus .....	53
Local EventBus .....	54
Cluster Map Interface .....	55
Requirements .....	55
Map creation .....	56
Map Changes .....	56
Map Destruction .....	57
5. Plugin Development .....	58

Writing Plugin Code .....	58
Using annotation support .....	59
Using older non-annotation based implementation .....	60
Implementation of processing method .....	60
Plugin Configuration .....	62
How Packets are Processed by the SM and Plugins .....	63
Introduction .....	63
SASL Custom Mechanisms and Configuration .....	67
Basic SASL Configuration .....	67
Logging/Authentication .....	69
Built-in Mechanisms .....	69
Custom Mechanisms Development .....	69
6. Using Maven .....	71
Setting up Maven in Windows .....	71
Requirements .....	71
Setting up Environment Variables .....	71
Testing Maven .....	73
A Very Short Maven Guide .....	74
Snapshot Compilation and Snapshot Package Generation .....	74
Release Compilation, Generation .....	74
Generating tar.gz, tar.bz2 File With Sources Only .....	74
Maven 2.x Support .....	74
7. Tests .....	76
Tests .....	76
Functional Tests .....	76
Performance Tests .....	78
Stability Tests .....	80
Tigase Test Suite .....	80
Running Tigase Test Suite (TTS) .....	80
Test Suite Scripting Language .....	82
Writing Tests for Plugins .....	83
Test Case Parameters Description .....	85
Test Report Configuration .....	85
Basic Test Parameters .....	86
Test Case Parameters .....	87
8. Experimental .....	90
Dynamic Rosters .....	90
Problem Description .....	90
Syntax and Semantics .....	90
Retrieving Contact Data .....	90
Updating/Saving Extra Information About the Contact .....	91
Configuration .....	92
Mobile Optimizations .....	92
Problem Description .....	92
Solution .....	92
Queuing Algorithms .....	93
Configuration .....	93
Bosh Session Cache .....	93
Problem Description .....	93
Bosh Session Cache Description .....	94
Cache Protocol .....	94
9. Old Stuff .....	96
10. Tigase DB Schema Explained .....	97
11. Why the most recent JDK? .....	99

12. Generating Tigase Installer .....	100
13. API Description for Virtual Domains Management in the Tigase Server .....	101
14. Stanza Limitations .....	103
Escape Characters .....	103

---

# Chapter 1. Basic Information

## Tigase Server Elements

To make it easier to get into the code below are defined basic terms in the Tigase server world and there is a brief explanation how the server is designed and implemented. This document also points you to basic interfaces and implementations which can be used as example code reference.

Logically all server code can be divided into 3 kinds of modules: **components**, **plug-ins** and **connectors**.

1. **Components** are the main element of Tigase server. Components are a bigger piece of code which can have separate address, receive and send stanzas, and be configured to respond to numerous events. Sample components implemented for Tigase server are: *c2s connection manager*, *s2s connection manager*, *session manager*, *XEP-0114 - external component connection manager*, *MUC - multi user char rooms*.
2. **Plug-ins** are usually small pieces of code responsible for processing specific XMPP stanzas. They don't have their own address. As a result of stanza processing they can produce new XMPP stanzas. Plug-ins are loaded by *session manager* component or the *c2s connection manager* component. Sample plug-ins are: *vCard stanza processing*, *jabber:iq:register* to register new user accounts, *presence stanza processing*, and *jabber:iq:auth* for non-sasl authentication.
3. **Connectors** are modules responsible for access to data repositories like databases or LDAP to store and retrieve user data. There are 2 kinds of connectors: authentication connectors and user data connectors. Both of them are independent and can connect to different data sources. Sample connectors are: *JDBC database connector*, *XMLDB - embedded database connector*, *Drupal database connector*, and the *LibreSource database connector*.

There is an API defined for each kind of above modules and all you have to do is enable the implementation of that specific interface. Then the module can be loaded to the server based on its configuration settings. There is also abstract classes available, implementing these interfaces to make development easier.

Here is a brief list of all interfaces to look at and for more details you have to refer to the guide for specific kind of module.

## Components

This is list of interfaces to look at when you work on a new component:

1. **tigase.server.ServerComponent** - This is the very basic interface for component. All components must implement it.
2. **tigase.server.MessageReceiver** - This interface extends `ServerComponent` and is required to implement by components which want to receive data packets like *session manager* and *c2s connection manager*.
3. **tigase.conf.Configurable** - Implementing this interface is required to make it configurable. For each object of this type, configuration is pushed to it at any time at runtime. This is necessary to make it possible to change configuration at runtime. Be careful to implement this properly as it can cause issues for modules that cannot be configured.
4. **tigase.disco.XMPPService** - Objects using this interface can respond to "ServiceDiscovery" requests.
5. **tigase.stats.StatisticsContainer** - Objects using this interface can return runtime statistics. Any object can collect job statistics and implementing this interface guarantees that statistics will be presented in consistent way to user who wants to see them.

Instead of implementing above interfaces directly I would recommend to extend one of existing abstract classes which take care of the most of "dirty and boring" stuff. Here is a list the most useful abstract classes:

- **tigase.server.AbstractMessageReceiver** - Implements 4 basic interfaces:

ServerComponent, MessageReceiver, Configurable and StatisticsContainer. AbstractMessageReceiver also manages internal data queues using it's own threads which prevents deadlocks from resource starvation. It offers even-driven data processing which means whenever packet arrives the abstract `void processPacket(Packet packet);` method is called to process it. You have to implement this abstract method in your component, if your component wants to send a packet (in response to data it received for example).

```
boolean addOutPacket(Packet packet)
```

- **tigase.server.ConnectionManager** - This is an extension of AbstractMessageReceiver abstract class. As the name says this class takes care of all network connection management stuff. If your component needs to send and receive data directly from the network (like c2s connection, s2s connection or external component) you should use this implementation as a basic class. It takes care of all things related to networking, I/O, reconnecting, listening on socket, connecting and so on. If you extend this class you have to expect data coming from two sources: from the MessageRouter and this is when the

```
abstract void processPacket(Packet packet);
```

method is called and from network connection and then the

```
abstract Queue processSocketData(XMPPIService serv);
```

method is called.

## Plug-ins

All Tigase plugins currently implemented are located in package: `tigase.xmpp.impl`. You can use this code as a sample code base. There are 3 types of plug-ins and they are defined in interfaces located in `tigase.xmpp` package:

1. **XMPPProcessorIfc** - The most important and basic plug-in. This is the most common plug-in type which just processes stanzas in normal mode. It receives packets, processes them on behalf of the user and returns resulting stanzas.
2. **XMPPPreprocessorIfc** - This plugin performs pre-processing of the packet, intended for the pre-processors to setup for packet blocking.
3. **XMPPPostprocessorIfc** - This plugin performs processing of packets for which there was no specific processor.

## Connector

### Data, Stanzas, Packets - Data Flow and Processing

Data received from the network are read from the network sockets as bytes by code in the `tigase.io` package. Bytes then are changed into characters in classes of `tigase.net` package and as characters they are sent to the XML parser (`tigase.xml`) which turns them to XML DOM structures.

All data inside the server is exchanged in XML DOM form as this is the format used by XMPP protocol. For basic XML data processing (parsing characters stream, building DOM, manipulate XML elements

and attributes) we use Tigase XML parser and DOM builder [<https://projects.tigase.org/projects/tigase-xmltools>].

Each stanza is stored in the `tigase.xml.Element` object. Every Element can contain any number of child Elements and any number of attributes. You can access all these data through the class API.

To simplify some, most common operations Element is wrapped in `tigase.server.Packet` class which offers another level of API for the most common operations like preparation of response stanza based on the element it contains (swap to/from values, put type=result attribute and others).



---

# Chapter 2. Hack Tigase XMPP Server in Eclipse

If you want to write code for **Tigase** server we recommend using Eclipse IDE [<https://eclipse.org/downloads/>]. Either the IDE for Java or Java EE developers will work.

## Requirements

Eclipse IDE currently requires the use of Java Runtime Environment 7 [<http://www.oracle.com/technetwork/java/javase/downloads/jre7-downloads-1880261.html>]. Although this is an outdated version of Java (and Tigase requires JDK version 8) Eclipse has not yet moved to the latest Java Build so you will need both versions installed.

You will also need the M2E plugin for Maven integration, however this can be done inside Elcipse now, so refer to the Plugin Installation section for that.

## Installation

Eclipse does not come as an installer, but rather an archive. Extract the directory to a working location wherever you would like. Now install the JRE 7 software, location is not important as Eclipse will find it autmoatically.

Before we begin, we will need to clone the repository from git.

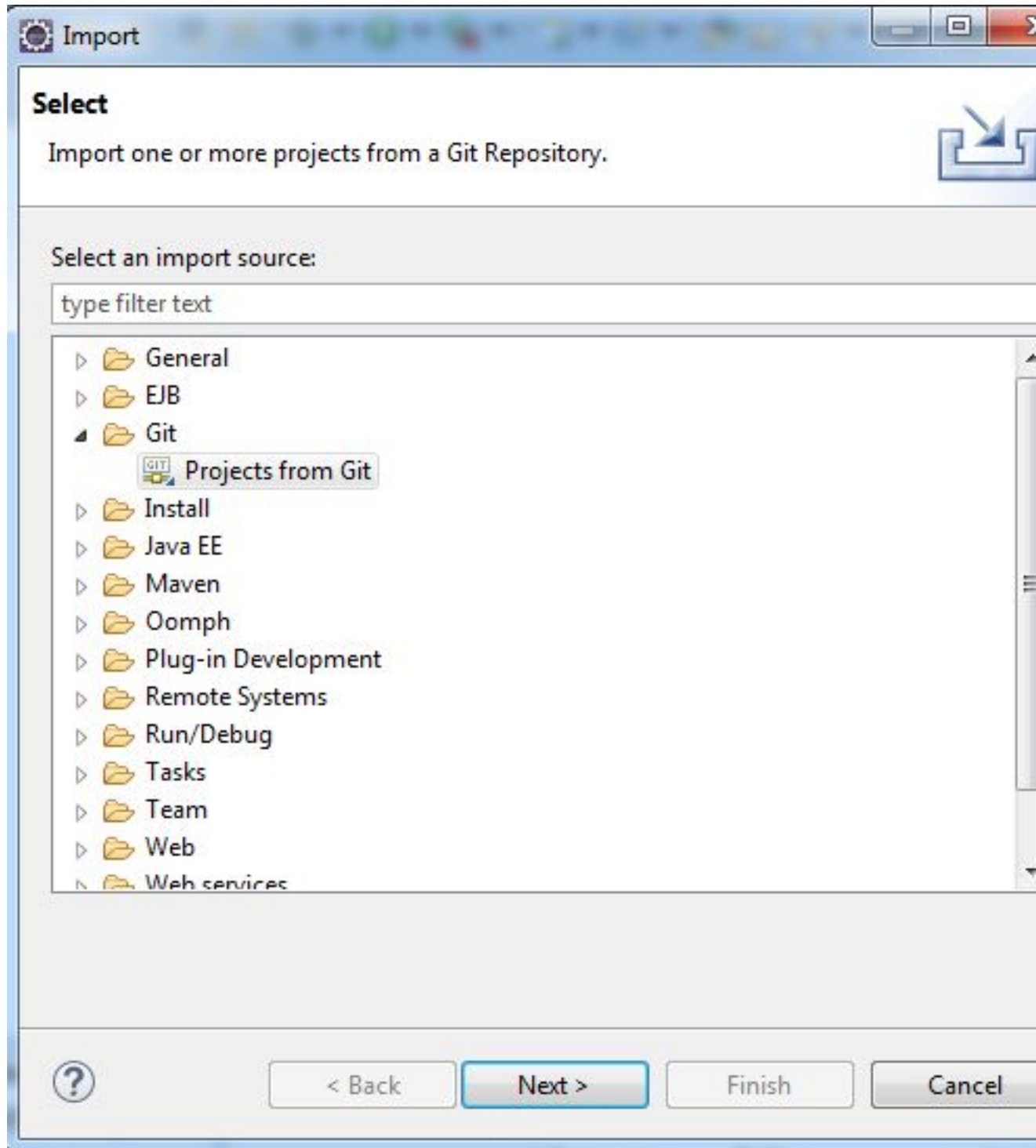
## Linux

For linux operating systems, navigate to a directory where you want the repository to be cloned to and type the following into terminal.

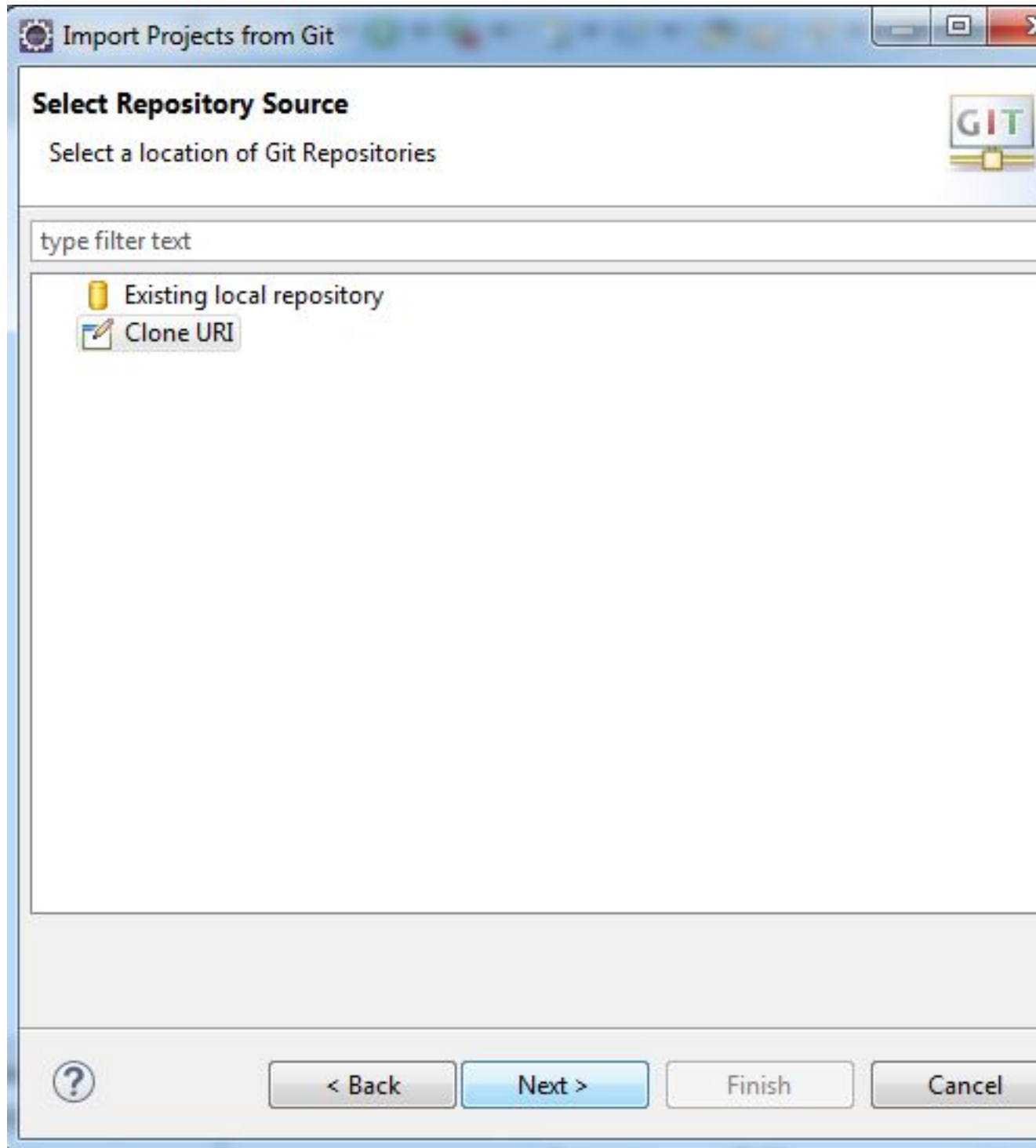
```
git clone https://repository.tigase.org/git/tigase-server.git
```

## Windows

Please see the Windows coding guide for instructions on how to obtain source code from git. If you don't want to install git software specifically, you can use Eclipse's git plugin to obtain the repository without any new software. First click on File, then Import... Next select from Git folder and the Projects from Git



Click next, and now select clone URI



Now click next, and in this window enter the following into the URI field

`git://repository.tigase.org/git/tigase-server.git`

The rest of the fields will populate automatically

**Import Projects from Git**

**Source Git Repository**

Enter the location of the source repository.

**Location**

URI:

Host:

Repository path:

**Connection**

Protocol:

Port:

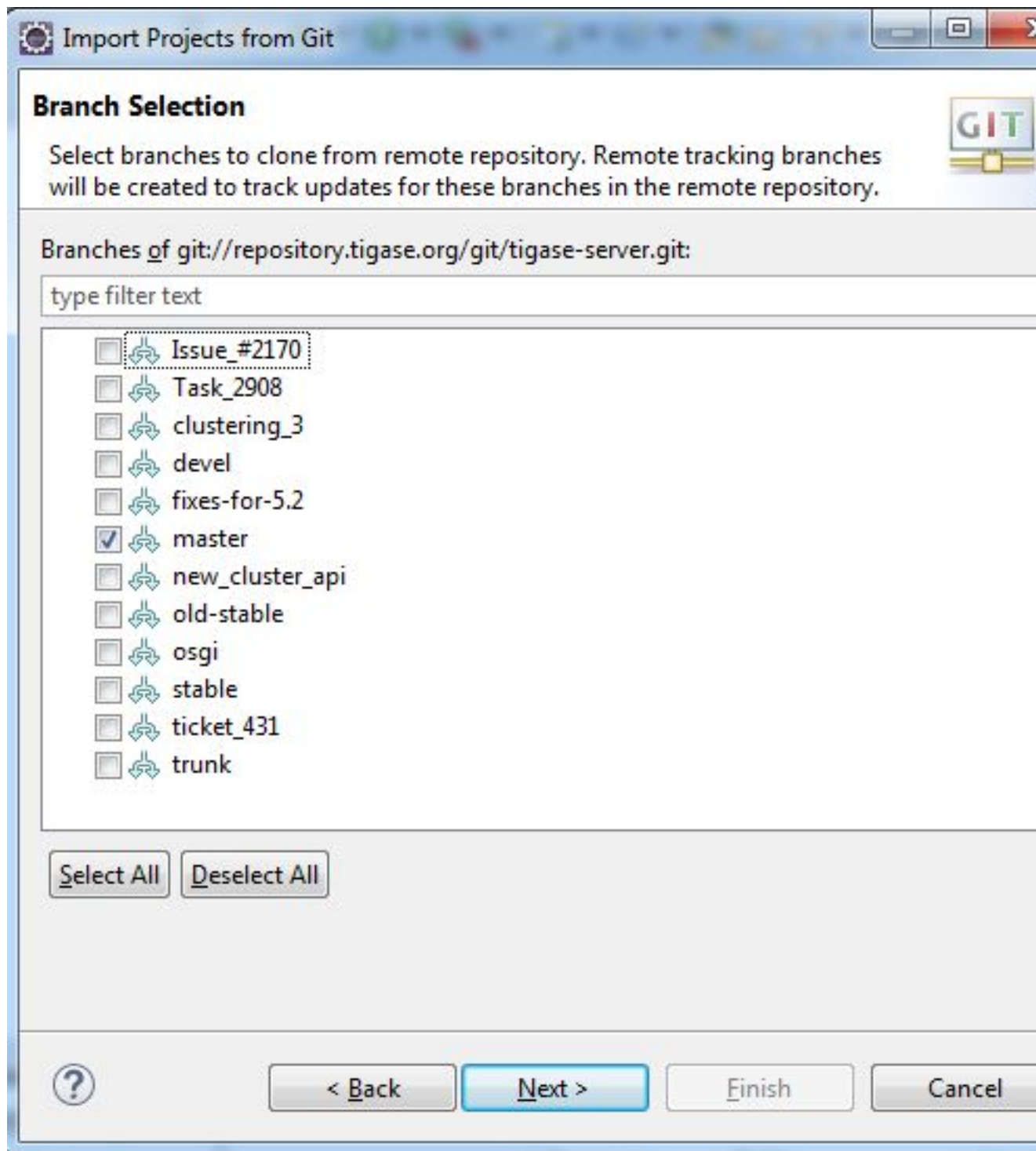
**Authentication**

User:

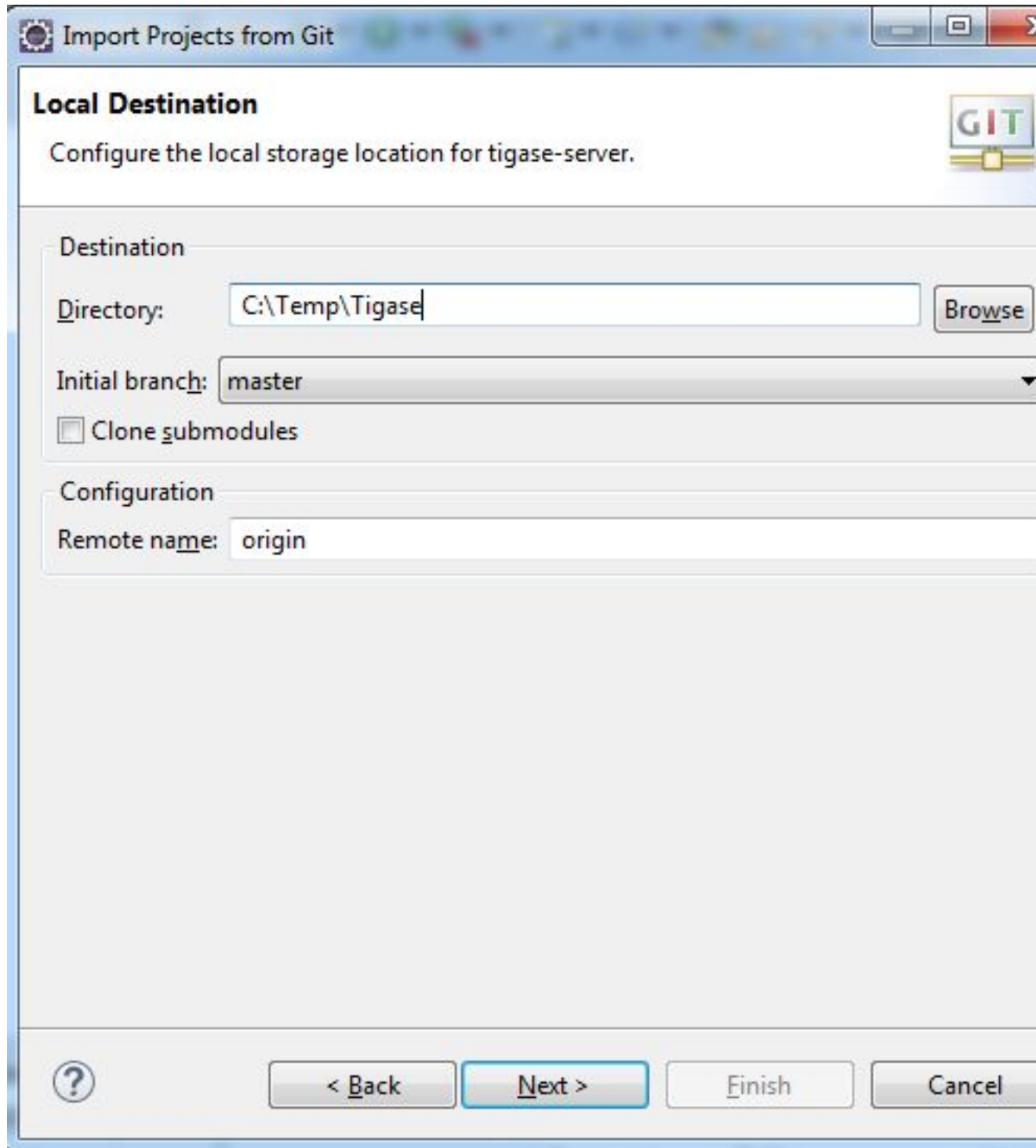
Password:

☐ Store in Secure Store

Select the master branch, and any branches you wish to edit. **The master branch should be the only one you need, branches are used for specific code changes**



Now select the directory where you wanted to clone the repository to. This was function as the project root directory you will use later on in the setup.



Once you click next Eclipse will download the repository and any branches you selected to that directory. Note you will be unable to import this git directory since there are no git a project specific files downloaded. However, once downloading is complete you may click cancel, and the git repository will remain in the directory you have chosen.

## Setup

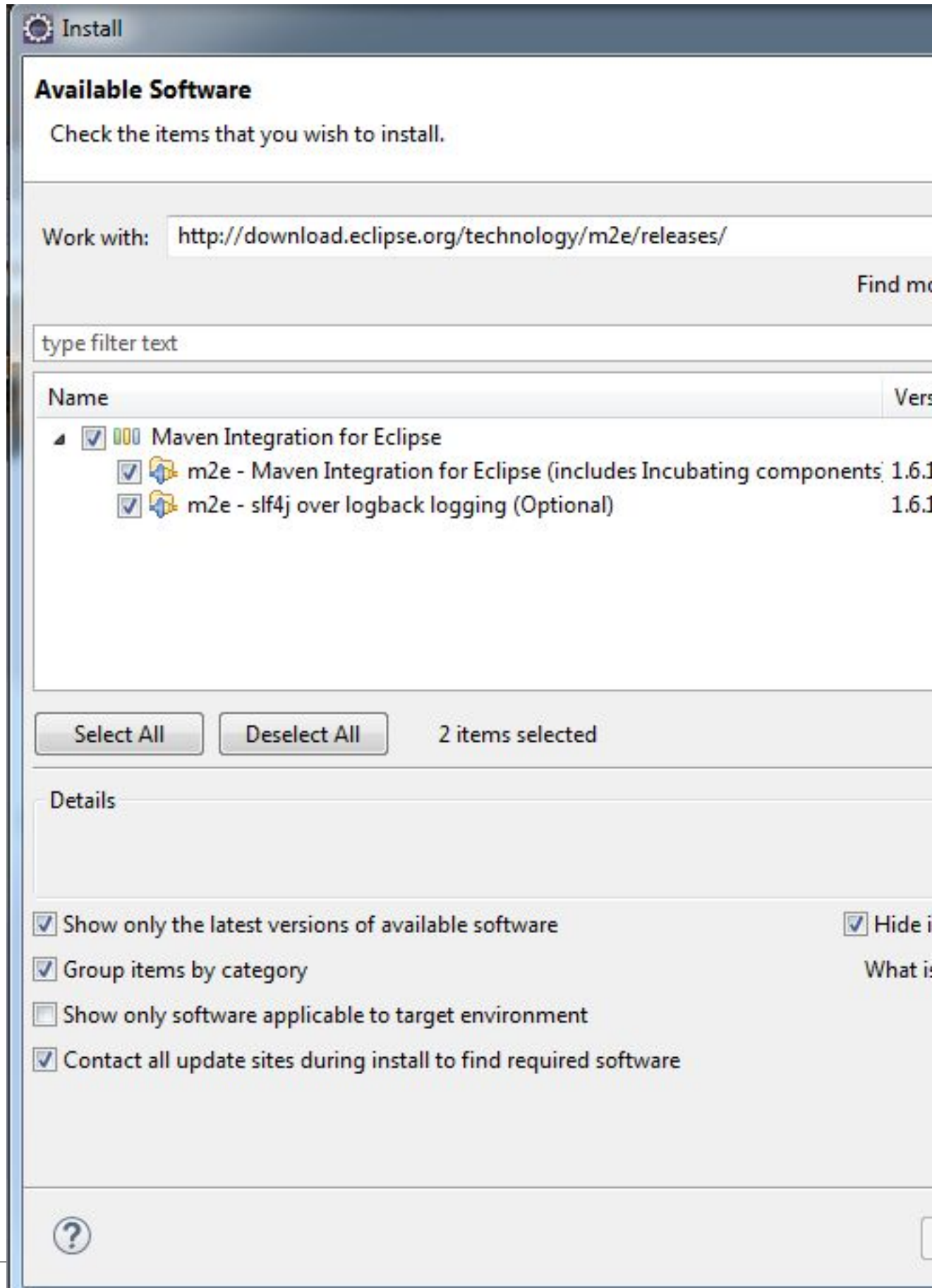
Once you have the main window open and have established a workspace (where most of your working files will be stored), click on Help and then Install New Software...



Under the Work With field enter the following and press enter: <http://download.eclipse.org/technology/m2e/releases/>

**Note:** You may wish to click the Add... button and add the above location as a permanent software location to keep the location in memory



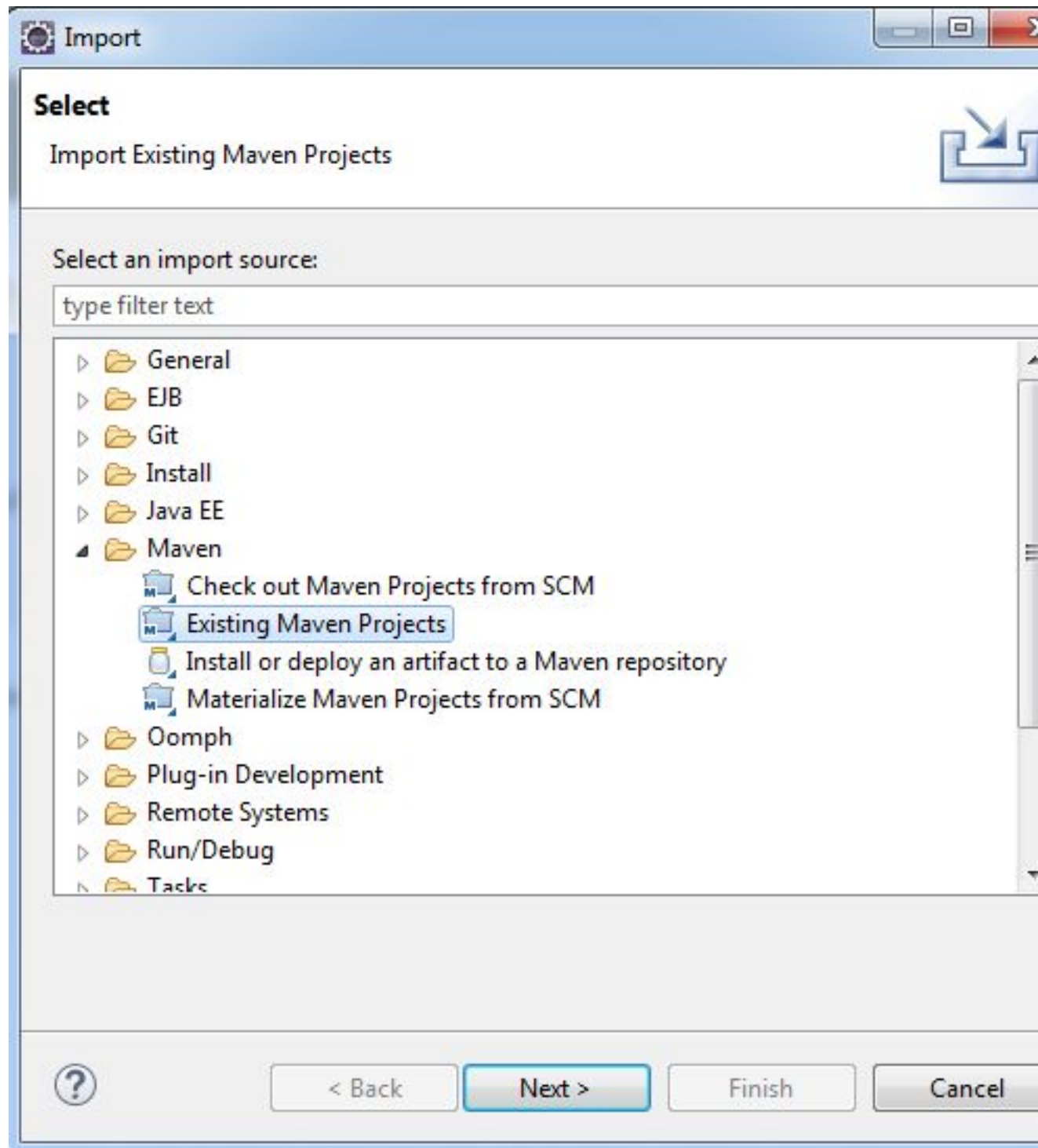




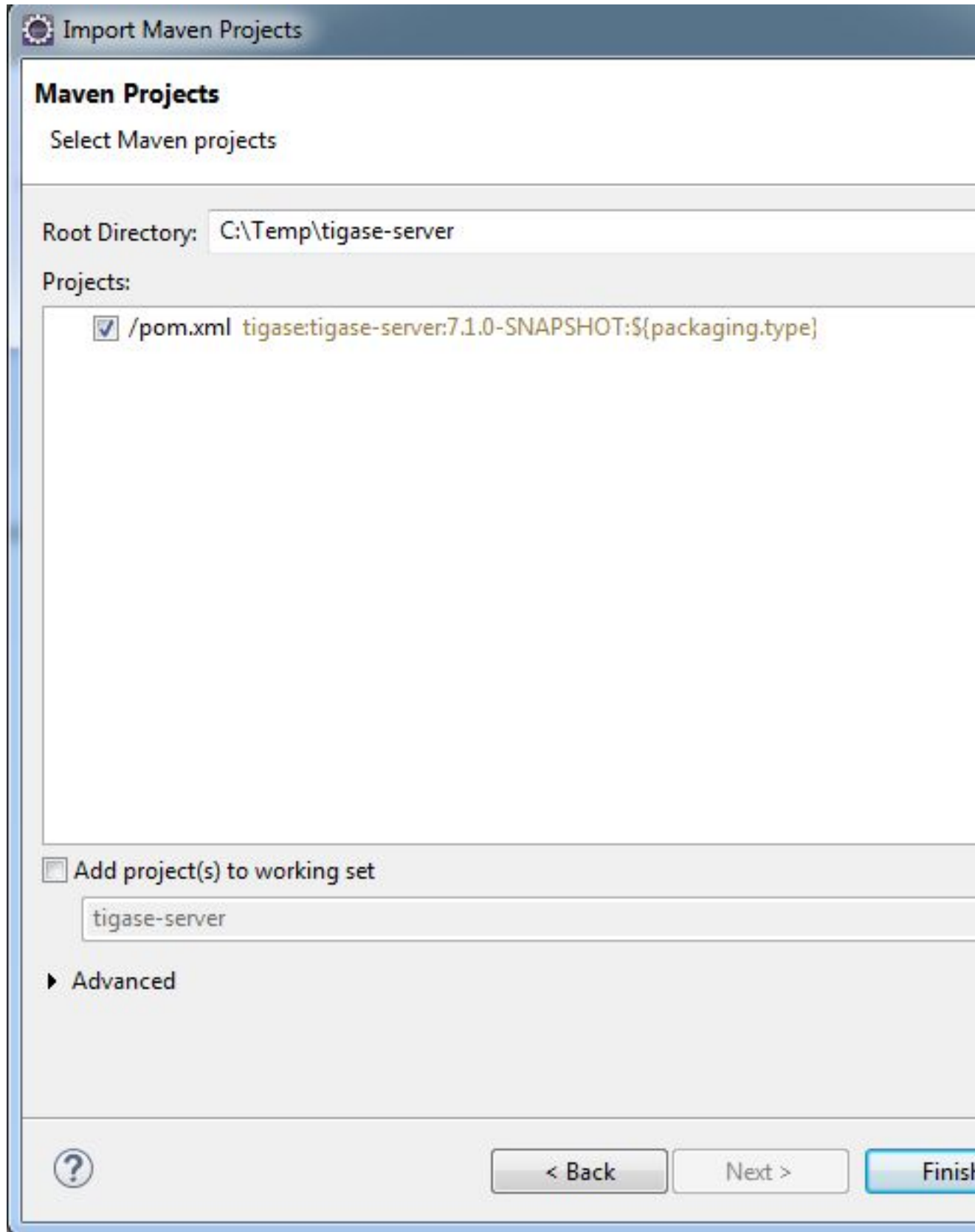
You should see the M2 Eclipse software packages show in the main window. Click the check-box and click Next. Once the installer is finished it will need to restart Eclipse.

Once that is done, lets connect Eclipse to the cloned repository.

Click File and Import... to bring up the import dialog window. Select Maven and then Existing Maven Project.



Now click Next and point the root directory to where you cloned the git repository, Eclipse should automatically see the pom.xml file and show up in the next window.



Once the import is finished, you are able to now begin working with Tigase's code inside Eclipse! Happy coding!

## API changes in the Tigase Server 5.x

The API changes can effect you only if you develop own code to run inside Tigase server. The changes are not extensive but in some circumstances may require many simple changes in a few files.

All the changes are related to introducing `tigase.xmpp.JID` and `tigase.xmpp.BareJID` classes. It is recommended to use them for all operations performed on the user JID instead of the `String` class which was used before changes.

There are a few advantages to using the new classes. First of all they do all the user JID checking and parsing, they also perform stringprep processing. Therefore if you use data kept by instance of the `JID` or `BareJID` you can be sure they are valid and correct.

These are not all advantages however. JID parsing code appears to use a lot of CPU power to conduct it's operations. JIDs and parts of the JIDs are used in many places of the stanza processing and the parsing is performed over and over again in all these places, wasting CPU cycles, memory and time. Therefore, great performance benefits can be gained from these new class are in if, once parsed, JIDs are reused in all further stanza processing.

This is where the `tigase.server.Packet` class comes in handy. Instances of the `Packet` class encloses XML stanza and pre-parses some, the most commonly used elements of the stanza, stanza source and destination addresses among them. As an effect there are all new methods available in the class:

```
JID getStanzaFrom();
JID getStanzaTo();
JID getFrom();
JID getTo();
JID getPacketFrom();
JID getPacketTo();
```

Whereas following methods are still available but have been deprecated:

```
String getElemFrom();
String getElemTo();
```

Please refer to the JavaDoc documentation for the `Packet` [<http://docs.tigase.org/tigase-server/snapshot/javadoc/tigase/server/Package.html>] class and methods to learn all the details of these methods and difference between them.

Another difference is that you can no longer create the `Packet` instance using a constructor. Instead there are a few factory methods available:

```
static Packet packetInstance(Element elem);
static Packet packetInstance(Element elem,
    JID stanzaFrom, JID stanzaTo);
```

Again, please refer to the JavaDoc documentation for all the details. The main point of using these methods is that they actually return an instance of one of the following classes instead of the `Packet` class: `Iq`, `Presence` or `Message`.

There is also a number of utility methods helping with creating a copy of the `Packet` instance preserving as much pre-parsed data as possible:

```
Packet copyElementOnly();
Packet errorResult(...);
Packet okResult(...);
Packet swapFromTo();
Packet swapStanzaFromTo();
```

We try to keep the JavaDoc [<http://docs.tigase.org/tigase-server/snapshot/javadoc/>] documentation as complete as possible. Please contact us if you find missing or incorrect information.

The main point is to reuse JID or BareJID instances in your code as much as possible. You never know, your code may run in highly loaded systems with throughput of 100k XMPP packets per second.

Another change. This one a bit risky as it is very difficult to find all places where this could be used. There are several utility classes and methods which accept source and destination address of a stanza and produce something. There was a great confusion with them, as in some of them the first was the source address and in others the destination address. All the code has been re-factored to keep the parameter order the same in all places. Right now the policy is: **source address first**. Therefore in all places where there was a method:

```
Packet method(String to, String from);
```

it has been changed to:

```
Packet method(JID from, JID to);
```

As far as I know most of these method were used only by myself so I do not expect much trouble for other developers.

---

# Chapter 3. Server Compilation

List of documents describing how to work with sources and how to compile them.

- Tigase XMPP Server 5.2.0 and Later - Compilation and Generating Distribution Packages
- Tigase Packages Dependency Change - Server Compilation Version 4.x or Later
- Server Compilation - Version 2.x and 3.x

## Tigase XMPP Server 5.2.0 and later - Compilation and Generating Distribution Packages

Starting with version 5.2.0 Tigase Server package distribution generation has switched from Ant to Maven. This will allow better dependency management as well as build repeatability.

For details on Maven and it's use, please see the Maven Guide.

## Distribution Packages

Starting from version 5.2.0 there will be two separate distribution archives:

- **-dist** is a minimal version containing only tigase-server, tigase-xmltools and tigase-utils
- **-dist-max** is a version containing all additional tigase components (MUC, PubSub, HTTP API, OSGi support, etc.) as well as dependencies required by those components.

They will be available as both zip and tarball.

## Building Server and Generating Packages

After cloning tigase-server repository:

```
git clone https://repository.tigase.org/git/tigase-server.git
cd tigase-server
```

You compile server with maven using project distribution profile (dist):

```
mvn --Pdist --f modules/master/pom.xml clean install
```

This will:

- compile server binaries
- generate javadoc documentation
- grab all latest versions of all declared dependencies and put them in jars/ directory
- create both types of distribution packages (-dist and -dist-max) and place them in pack/ directory

In order to create installer packages you have to execute two shell scripts:

```
./scripts/installer-prepare.sh
```

```
./scripts/installer-generate.sh
```

However, in order for them to succeed you have to build the server first using maven as described earlier. You should also have git, python2, docutils and LaTeX distributions installed (please see `src/main/izpack/README.txt` for details).

## Running Server

Afterwards you can run the server with the regular shell script:

```
./scripts/tigase.sh start etc/tigase.conf
```

Please bear in mind, that you need to provide correct setup in `etc/init.properties` configuration files for the server to work correctly.

## Tigase Packages Dependency Change - Server Compilation Version 4.x or Later

The dependency for Tigase Utils Package [<https://projects.tigase.org/projects/tigase-utils>] has changed. This is important for everybody who builds the Tigase server manually from sources using Ant [<http://ant.apache.org/>] tool. The Maven [<http://maven.apache.org/>] handles all the dependencies automatically and scripts have been updated.

Please keep reading for more details how to compile the server from sources in current repositories.

If you have an old **Tigase MUC** or **Tigase Extras** package lying in the `server/libs/` directory please remove it now. You have to update it too and copy it over to the `server/jars/` directory after you completed steps below.

For all those who build the server from sources manually using Ant [<http://ant.apache.org/>] here is a short guide:

1. Checkout all the sources first:
  - <https://projects.tigase.org/projects/tigase-xmltools/repository>
  - <https://projects.tigase.org/projects/tigase-utils/repository>
  - <https://projects.tigase.org/projects/tigase-server/repository>
2. Build the Tigase XMLTools and copy the jar file over to the **utils** and **\*server\*** `libs/` directory
  - `cd xmltools`
  - `ant clean jar-dist`
  - `cp jars/tigase-xmltools.jar ../utils/libs`
  - `cp jars/tigase-xmltools.jar ../server/libs`
3. Build the Tigase Utils and copy the jar file to the **server** `libs/` directory
  - `cd ../utils`
  - `ant clean jar-dist`

- `cp jars/tigase-utils.jar ../server/libs`

#### 4. Build the Tigase Server binary

- `cd ../server`
- `ant clean jar-dist`

This is a very short guide but I hope it helps. If you have any problems, please let me know.

Addendum: starting with version 5.2.0 all libraries and jar files for the server are in jars/ directory; however with that version we strongly encourage to switch to maven build system as we are phasing out Ant - please follow guide [Tigase XMPP Server 5.2.0 and Later - Compilation and Generating Distribution Packages](#)



---

# Chapter 4. Component Development

A component in the Tigase is an entity with its own JID address. It can receive packets, process them, and can also generate packets.

An example of the best known components is MUC or PubSub. In Tigase however, almost everything is actually a component: Session Manager, s2s connections manager, Message Router, etc... Components are loaded based on the server configuration, new components can be loaded and activated at run-time. You can easily replace a component implementation and the only change to make is a class name in the configuration entry.

Creating components for Tigase server is an essential part of the server development hence there is a lot of useful API and ready to use code available. This guide should help you to get familiar with the API and how to quickly and efficiently create your own component implementations.

1. Component implementation - Lesson 1 - Basics
2. Component implementation - Lesson 2 - Configuration
3. Component implementation - Lesson 3 - Multi-Threading
4. Component implementation - Lesson 4 - Service Discovery
5. Component implementation - Lesson 5 - Statistics
6. Component implementation - Lesson 6 - Scripting Support
7. Component implementation - Lesson 7 - Data Repository
8. Component implementation - Lesson 8 - Startup Time
9. Configuration API
10. Packet Filtering in Component

## Component Implementation - Lesson 1 - Basics

Creating a Tigase component is actually very simple and with broad API available you can create a powerful component with just a few lines of code. You can find detailed API description elsewhere. This series presents hands on lessons with code examples, teaching how to get desired results in the simplest possible code using existing Tigase API.

Even though all Tigase components are just implementations of the **ServerComponent** interface I will keep such a low level information to necessary minimum. Creating a new component based on just interfaces, while very possible, is not very effective. This guide intends to teach you how to make use of what is already there, ready to use with a minimal coding effort.

This is just the first lesson of the series where I cover basics of the component implementation.

Let's get started and create the Tigase component:

```
import java.util.logging.Logger;
```

```
import tigase.server.AbstractMessageReceiver;
import tigase.server.Packet;

public class TestComponent extends AbstractMessageReceiver {

    private static final Logger log = Logger.getLogger(TestComponent.class.getName());

    @Override
    public void processPacket(Packet packet) {
        log.finest("My packet: -" + packet.toString());
    }
}
```

The only element mandatory when you extend **AbstractMessageReceiver** is the implementation of **void processPacket(Packet packet)** method. This is actually logical as the main task for your component is processing packets. Class name for our new component is **TestComponent** and we have also initialized a separated logger for this class. Doing This is very useful as it allows us to easily find log entries created by our class.

With these a few lines of code you have a fully functional Tigase component which can be loaded to the Tigase server; it can receive and process packets, shows as an element on service discovery list (for administrators only), responds to administrator ad-hoc commands, supports scripting, generates statistics, can be deployed as an external component, and a few other things.

Before we go any further with the implementation let's configure the component in Tigase server so it is loaded next time the server starts. Assuming our **init.properties** file looks like this one:

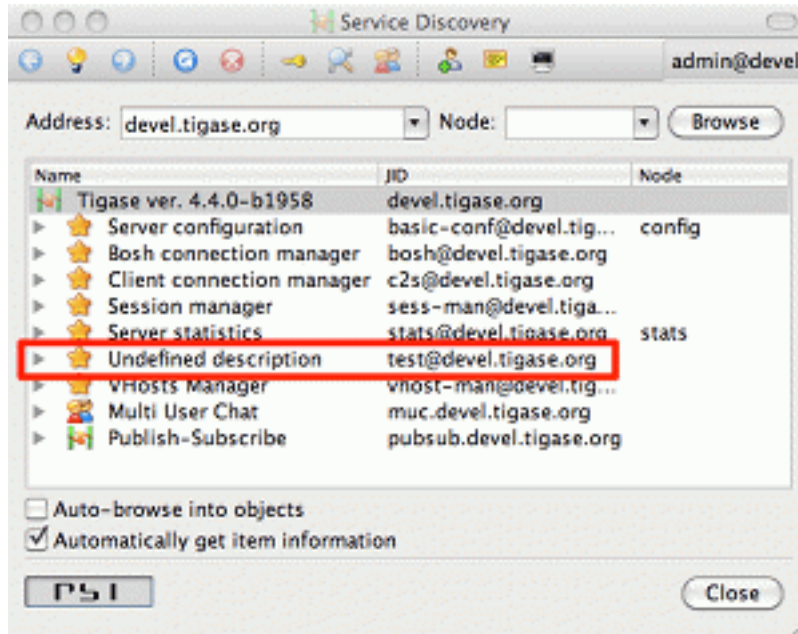
```
config-type = ---gen-config-def
--debug = server
--user-db = derby
--admins = admin@devel.tigase.org
--user-db-uri = jdbc:derby:/Tigase/tigasedb
--virt-hosts = devel.tigase.org
--comp-name-1 = muc
--comp-class-1 = tigase.muc.MUCComponent
--comp-name-2 = pubsub
--comp-class-2 = tigase.pubsub.PubSubComponent
```

We can see that it already is configured to load two other components: **MUC** and **PubSub**. Let's add a third - our new component to the configuration file by appending two following lines in the properties file:

```
--comp-name-3 = test
--comp-class-3 = TestComponent
```

Now we have to remove the **etc/tigase.xml** file and restart the server.

There are a few ways to check whether our component has been loaded to the server. Probably the easiest is to connect to the server from an administrator account and look at the service discovery list.



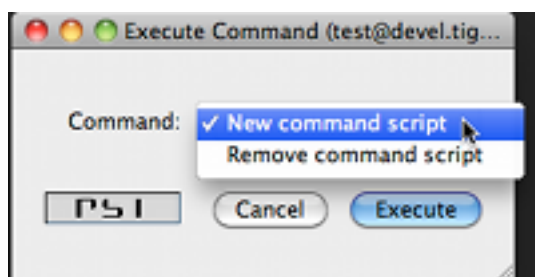
If everything goes well you should see an entry on the list similar to the highlighted one on the screenshot. The component description is "Undefined description" which is a default description and we can change it later on, the component default JID is: **test@devel.tigase.org**, where **devel.tigase.org** is the server domain and test is the component name.

Another way to find out if the component has been loaded is by looking at the log files. Getting yourself familiar with Tigase log files will be very useful thing if you plan on developing Tigase components. So let's look at the log file **logs/tigase.log.0**, if the component has been loaded you should find following lines in the log:

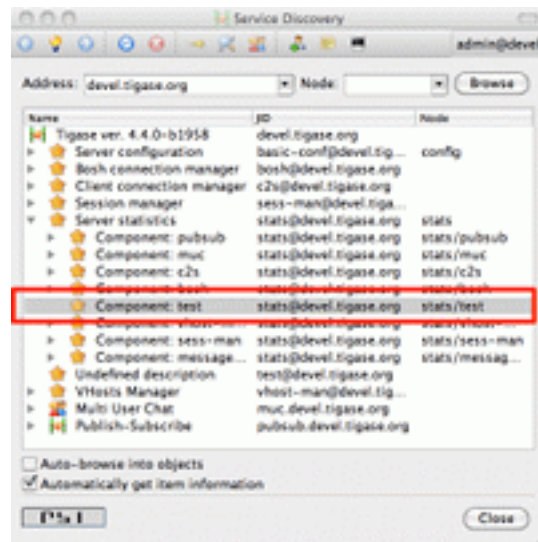
```
MessageRouter.setProperties() FINER: Loading and registering message receiver: test
MessageRouter.addRouter() INFO: Adding receiver: TestComponent
MessageRouter.addComponent() INFO: Adding component: TestComponent
MessageRouter.addComponent() FINER: Adding: test component to basic-conf registrat
Configurator.componentAdded() CONFIG: component: test
```

If your component did not load you should first check configuration files. Maybe you forgot to remove the **tigase.xml** file before restarting the server or alternatively the Tigase could not find your class at startup time. Make sure your class is in **CLASSPATH** or copy a JAR file with your class to Tigase **libs/** directory.

Assuming everything went well and your component is loaded by the sever and it shows on the service discovery list as on the screenshot above you can double click on it to get a window with a list of ad-hoc commands - administrator scripts. A window on the screenshot shows only two basic commands for adding and removing script which is a good start.



Moreover, you can browse the server statistics in the service discovery window to find your new test component on the list. If you click on the component it shows you a window with component statistics, very basic packets counters.



As we can see with just a few lines of code our new component is quite mighty and can do a lot of things without much effort from the developer side.

Now, the time has come to the most important question. Can our new component do something useful, that is can it receive and process XMPP packets?

Let's try it out. Using your favorite client send a message to JID: **test@devel.tigase.org** (assuming your server is configured for **devel.tigase.org** domain). You can either use kind of XML console in your client or just send a plain message to the component JID. According to our code in **processPacket(...)** method it should log our message. For this test I have sent a message with subject: *"test message"* and body: *"this is a test"*. The log file should contain following entry:

```
TestComponent.processPacket() FINEST: My packet: to=null, from=null,
data=<message from="admin@devel.tigase.org/devel"
  to="test@devel.tigase.org" id="abcaa" xmlns="jabber:client">
  <subject>test message</subject>
  <body>this is a test</body>
</message>, XMLNS=jabber:client, priority=NORMAL
```

If this is a case we can be sure that everything works as expected and all we now have to do is to fill the **processPacket(...)** method with some useful code.

## Component Implementation - Lesson 2 - Configuration

It might be hard to tell what the first important thing you should do with your new component implementation. Different developers may have a different view on this. It seems to me however that it is always a good idea to give to your component a way to configure it and provide some runtime settings.

This guide describes how to add configuration handling to your component. There is detailed Configuration API description available so again I am not getting deep into all details just the necessary code.

To demonstrate how to implement component configuration let's say we want to configure which types of packets will be logged by the component. There are three possible packet types: **message**, **presence** and **iq** and we want to be able to configure logging of any combination of the three. Furthermore we also want to be able to configure the text which is prepended to the logged message and to optionally switch secure login. (Secure logging replaces all packet CData with text: *CData size: NN* to protect user privacy.)

Let's create the following private variables in our component:

```
private String[] packetTypes = {"message", "-presence", "-iq"};
private String prependText = "-My packet: -";
private boolean secureLogging = false;
```

As the component configuration is maintained in a form of a (*key*, *value*) map, we have to invent keys for each of our configuration entry:

```
private static final String PACKET_TYPES_KEY = "-packet-types";
private static final String PREPEND_TEXT_KEY = "-log-prepend";
private static final String SECURE_LOGGING_KEY = "-secure-logging";
```

There are two methods used to maintain the component configuration: `getDefaults(...)` where the component provides some configuration defaults and `setProperties(...)` which sets a working configuration for the component:

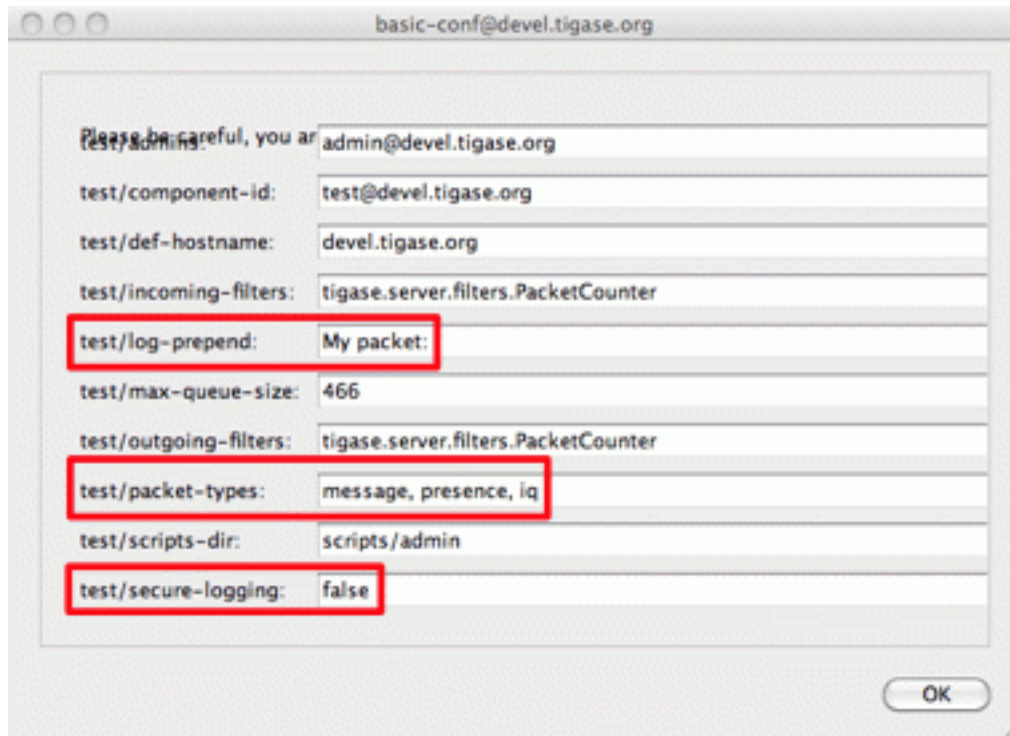
```
@Override
public Map<String, Object> getDefaults(Map<String, Object> params) {
    Map<String, Object> defs = super.getDefaults(params);
    defs.put(PACKET_TYPES_KEY, packetTypes);
    defs.put(PREPEND_TEXT_KEY, prependText);
    defs.put(SECURE_LOGGING_KEY, secureLogging);
    return defs;
}

@Override
public void setProperties(Map<String, Object> props) {
    super.setProperties(props);
    if (props.get( PACKET_TYPES_KEY ) != null ) {
        packetTypes = (String[]) props.get( PACKET_TYPES_KEY );
    }
    if (props.get( PREPEND_TEXT_KEY ) != null ) {
        prependText = (String) props.get( PREPEND_TEXT_KEY );
    }
    if (props.get( SECURE_LOGGING_KEY ) != null ) {
        secureLogging = (Boolean) props.get( SECURE_LOGGING_KEY );
    }
}
```

You do not have to implement the `getDefaults(...)` method and provide default settings for your configuration, but doing so gives you a few benefits.

The first from a developer point of view, you don't have to check in the `setProperties(...)` whether the value is of a correct type or convert it from `String` to the correct type as it always be either the default or user provided. It will be of a correct type as the configuration framework takes care of the types comparing between the user provided settings and default values. So this just makes your `setProperties(...)` code much simpler and clearer.

Please note that currently Tigase allows changing properties automatically. Hence you should check each time if a given property was updated at the given call of `setProperties()`.



Secondly this also makes the administrator's life easier. As you can see on the screenshot, configuration parameters provided with default values can be changed via configuration ad-hoc commands. So the administrator can maintain your component during run-time from his XMPP client.

Regardless, if you implemented the `getDefaults(...)` method or not you can always manually add parameters to the `init.properties` file.

The syntax in `init.properties` file is very simple and is described in details in the *Admin Guide*. As it shows on the screenshot the configuration parameter name consists of: *component name, property key*. To set the configuration for your component in `init.properties` file you have to append following lines to the file:

```
test/log-prepend="My packet: -"
test/packet-types[s]=message,presence,iq
test/secure-logging[B]=true
```

The square brackets denote the property type, have a look at the *Admin Guide* documentation for more details.

And this is the complete code of the new component with a modified `processPacket(...)` method taking advantage of configuration settings:

```
import java.util.Map;
import java.util.logging.Logger;
import tigase.server.AbstractMessageReceiver;
import tigase.server.Packet;
```

```
public class TestComponent extends AbstractMessageReceiver {

    private static final Logger log =
        Logger.getLogger(TestComponent.class.getName());

    private static final String PACKET_TYPES_KEY = "-packet-types";
    private static final String PREPEND_TEXT_KEY = "-log-prepend";
    private static final String SECURE_LOGGING_KEY = "-secure-logging";

    private String[] packetTypes = {"message", "-presence", "-iq"};
    private String prependText = "-My packet: -";
    private boolean secureLogging = false;

    @Override
    public void processPacket(Packet packet) {
        for (String pType -: packetTypes) {
            if (pType == packet.getElemName()) {
                log.finest(prependText + packet.toString(secureLogging));
            }
        }
    }

    @Override
    public Map<String, Object> getDefaults(Map<String, Object> params) {
        Map<String, Object> defs = super.getDefaults(params);
        defs.put(PACKET_TYPES_KEY, packetTypes);
        defs.put(PREPEND_TEXT_KEY, prependText);
        defs.put(SECURE_LOGGING_KEY, secureLogging);
        return defs;
    }

    @Override
    public void setProperties(Map<String, Object> props) {
        super.setProperties(props);
        if (props.get( PACKET_TYPES_KEY -) != null -) {
            packetTypes = (String[]) props.get( PACKET_TYPES_KEY -);
        }
        // Make sure we can compare element names by reference
        // instead of String content
        for (int i = 0; i < packetTypes.length; i++) {
            packetTypes[i] = packetTypes[i].intern();
        }
        if (props.get( PREPEND_TEXT_KEY -) != null -) {
            prependText = (String) props.get( PREPEND_TEXT_KEY -);
        }
        if (props.get( SECURE_LOGGING_KEY -) != null -) {
            secureLogging = (Boolean) props.get( SECURE_LOGGING_KEY -);
        }
    }
}
```

Of course we can do much more useful packet processing in the `processPacket(...)` method. This is just an example code. Please note: comparing packet element name with our packet type by reference

is intentional and allowed in this context. All **Element** names are processed with `String.intern()` function to preserve memory and improve performance of string comparison.

## Component Implementation - Lesson 3 - Multi-Threading

Multi core and multi CPU machines very common nowadays, especially for an application like the XMPP server you most likely deployed your service on. Your new custom component however, processes all packets in a single thread.

This is especially important if the packet processing is CPU expensive like, for example, SPAM checking. In such a case you could experience single Core/CPU usage at 100% while other Cores/CPU's are idling. Ideally, you want your component to use all available CPU's.

Tigase API offers a very simple way to execute component's `processPacket(Packet packet)` method in multiple threads. Methods `int processingOutThreads()` and `int processingInThreads()` returns number of threads assigned to the component. By default it returns just `1` as not all component implementations are prepared to process packets concurrently. By overwriting the method you can return any value you think is appropriate for the implementation. Please note, there are two methods, one is for a number of threads for incoming packets to the component and another for outgoing packets from the component. It used to be a single method but different components have different needs and the best performance can be achieved when the outgoing queues have a separate threads pool from incoming queues. Also some components only receive packets while other only send, therefore assigning an equal number of threads for both that could be a waste of resources.

If the packet processing is CPU bound only, you normally want to have as many threads as there are CPU's available:

```
@Override
public int processingInThreads() {
    return Runtime.getRuntime().availableProcessors();
}
@Override
public int processingOutThreads() {
    return Runtime.getRuntime().availableProcessors();
}
```

If the processing is I/O bound (network or database) you probably want to have more threads to process requests. It is hard to guess the ideal number of threads right on the first try. Instead you should run a few tests to see how many threads is best for implementation of the component.

Now you have many threads for processing your packets, but there is one slight problem with this. In many cases packet order is essential. If our `processPacket(...)` method is executed concurrently by a few threads it is quite possible that a message sent to user can takeover the message sent earlier. Especially if the first message was large and the second was small. We can prevent this by adjusting the method responsible for packet distribution among threads.

The algorithm for packets distribution among threads is very simple:

```
int thread_idx = hashCodeForPacket(packet) % threads_total;
```

So the key here is using the `hashCodeForPacket(...)` method. By overwriting it we can make sure that all packets addressed to the same user will always be processed by the same thread:



```
@Override
public int hashCodeForPacket(Packet packet) {
    if (packet.getElemTo() != null) {
        return packet.getElemTo().hashCode();
    }
    // This should not happen, every packet must have a destination
    // address, but maybe our SPAM checker is used for checking
    // strange kind of packets too....
    if (packet.getElemFrom() != null) {
        return packet.getElemFrom().hashCode();
    }
    // If this really happens on your system you should look
    // carefully at packets arriving to your component and
    // find a better way to calculate hashCode
    return 1;
}
```

The above two methods give control over the number of threads assigned to the packets processing in your component and to the packet distribution among threads. This is not all Tigase API has to offer in terms of multi-threading.

Sometimes you want to perform some periodic actions. You can of course create Timer instance and load it with TimerTasks. As there might be a need for this, every level of the Class hierarchy could end-up with multiple Timer (threads in fact) objects doing similar job and using resources. There are a few methods which allow you to reuse common Timer object to perform all sorts of actions.

First, you have three methods allowing your to perform some periodic actions:

```
public synchronized void everySecond();
public synchronized void everyMinute();
public synchronized void everyHour();
```

An example implementation for periodic notifications sent to some address could look like this one:

```
@Override
public synchronized void everyMinute() {
    super.everyMinute();
    if ((++delayCounter) >= notificationFrequency) {
        addOutPacket(Packet.getMessage(abuseAddress, getId(),
            StanzaType.chat, "-Detected spam messages: -" + spamCounter,
            "-Spam counter", null, newPacketId("spam-")));
        delayCounter = 0;
        spamCounter = 0;
    }
}
```

This method sends every **notificationFrequency** minute a message to **abuseAddress** reporting how many spam messages have been detected during last period. Please note, you have to call `super.everyMinute()` to make sure other actions are executed as well and you have to also remember to keep processing in this method to minimum, especially if you overwrite `everySecond()` method.

There are also two methods which allow you to schedule tasks executed at certain time, they are very similar to the `java.util.Timer` API with the only difference is that Timer is reused among all levels of Class hierarchy. There is a separate Timer for each Class instance though, to avoid interferences between separate components:

```
addTimerTask(TimerTask task, long delay, TimeUnit unit);
addTimerTask(TimerTask task, long delay);
```

There is one more method which can be used which is not directly related to multi-threading, but might be very helpful for executing some actions at a very specific point of time. This is the point of time when the server has just been initialized, that is all components have been created and received their configuration for the first time. When this happens Tigase calls `void initializationCompleted()` method for each server component. You can overwrite this method to execute some actions at the time when you are sure the Tigase has started and is fully functional.

Here is a code of an example component which uses all the API discussed in this article:

```
import java.util.Arrays;
import java.util.Map;
import java.util.logging.Logger;
import tigase.server.AbstractMessageReceiver;
import tigase.server.Packet;
import tigase.util.JIDUtils;
import tigase.xmpp.StanzaType;

public class TestComponent extends AbstractMessageReceiver {

    private static final Logger log =
        Logger.getLogger(TestComponent.class.getName());

    private static final String BAD_WORDS_KEY = "-bad-words";
    private static final String WHITELIST_KEY = "-white-list";
    private static final String PREPEND_TEXT_KEY = "-log-prepend";
    private static final String SECURE_LOGGING_KEY = "-secure-logging";
    private static final String ABUSE_ADDRESS_KEY = "-abuse-address";
    private static final String NOTIFICATION_FREQ_KEY = "-notification-freq";

    private String[] badWords = {"word1", "-word2", "-word3"};
    private String[] whiteList = {"admin@localhost"};
    private String prependText = "-Spam detected: -";
    private String abuseAddress = "-abuse@localhost";
    private int notificationFrequency = 10;
    private int delayCounter = 0;
    private boolean secureLogging = false;
    private long spamCounter = 0;

    @Override
    public void processPacket(Packet packet) {
        // Is this packet a message?
        if ("message" == packet.getElemName()) {
            String from = JIDUtils.getNodeID(packet.getElemFrom());
            // Is sender on the whitelist?
            if (Arrays.binarySearch(whiteList, from) < 0) {
                // The sender is not on whitelist so let's check the content
                String body = packet.getElemCDATA("/message/body");
                if (body != null && !body.isEmpty()) {
                    body = body.toLowerCase();
                    for (String word : badWords) {
                        if (body.contains(word)) {
                            log.finest(prependText + packet.toString(secureLogging));
                        }
                    }
                }
            }
        }
    }
}
```

```

        ++spamCounter;
        return;
    -}
-}
-}
-}
-}
-// Not a SPAM, return it for further processing
Packet result = packet.swapFromTo();
addOutPacket(result);
-}

@Override
public int processingInThreads() {
    return Runtime.getRuntime().availableProcessors();
-}

@Override
public int processingOutThreads() {
    return Runtime.getRuntime().availableProcessors();
-}

@Override
public int hashCodeForPacket(Packet packet) {
    if (packet.getElemTo() != null) {
        return packet.getElemTo().hashCode();
    -}
    -// This should not happen, every packet must have a destination
    -// address, but maybe our SPAM checker is used for checking
    -// strange kind of packets too....
    if (packet.getElemFrom() != null) {
        return packet.getElemFrom().hashCode();
    -}
    -// If this really happens on your system you should look carefully
    -// at packets arriving to your component and decide a better way
    -// to calculate hashCode
    return 1;
-}

@Override
public Map<String, Object> getDefaults(Map<String, Object> params) {
    Map<String, Object> defs = super.getDefaults(params);
    defs.put(BAD_WORDS_KEY, badWords);
    defs.put(WHITELIST_KEY, whiteList);
    defs.put(PREPEND_TEXT_KEY, prependText);
    defs.put(SECURE_LOGGING_KEY, secureLogging);
    defs.put(ABUSE_ADDRESS_KEY, abuseAddress);
    defs.put(NOTIFICATION_FREQ_KEY, notificationFrequency);
    return defs;
-}

@Override
public void setProperties(Map<String, Object> props) {
    super.setProperties(props);

```

```
badWords = (String[])props.get(BAD_WORDS_KEY);
whiteList = (String[])props.get(WHITELIST_KEY);
Arrays.sort(whiteList);
prependText = (String)props.get(PREPEND_TEXT_KEY);
secureLogging = (Boolean)props.get(SECURE_LOGGING_KEY);
abuseAddress = (String)props.get(ABUSE_ADDRESS_KEY);
notificationFrequency = (Integer)props.get(NOTIFICATION_FREQ_KEY);
-}

@Override
public synchronized void everyMinute() {
    super.everyMinute();
    if ((++delayCounter) >= notificationFrequency) {
        addOutPacket(Packet.getMessage(abuseAddress, getComponentId(),
            StanzaType.chat, "-Detected spam messages: -" + spamCounter,
            "-Spam counter", null, newPacketId("spam-")));
        delayCounter = 0;
        spamCounter = 0;
    }
-}
}
```

## Component Implementation - Lesson 4 - Service Discovery

You component still shows in the service discovery list as an element with "*Undefined description*". It also doesn't provide any interesting features or sub-nodes.

In this article I will show how to, in a simple way, change the basic component information presented on the service discovery list and how to add some service disco features. As a bit more advanced feature the guide will teach you about adding/removing service discovery nodes at run-time and about updating existing elements.

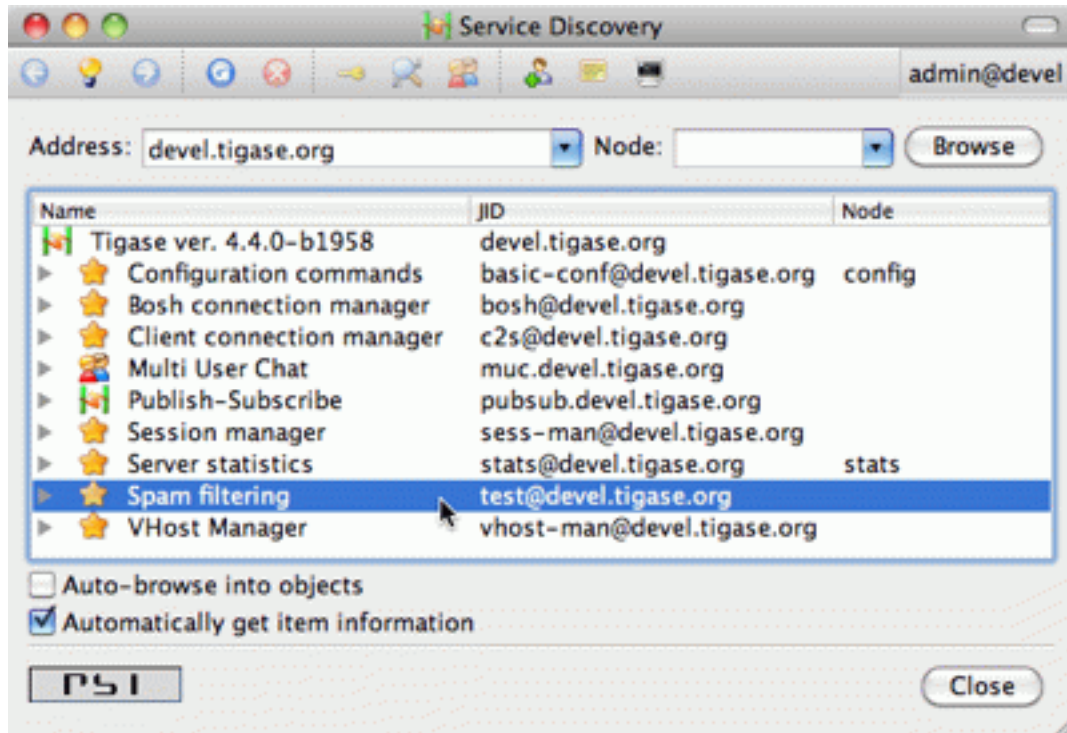
Component description and category type can be changed by overriding two following methods:

```
@Override
public String getDiscoDescription() {
    return "-Spam filtering";
}

@Override
public String getDiscoCategoryType() {
    return "-spam";
}
```

Please note, there is no such '**spam**' category type defined in the Service Discovery Identities registry [<http://xmpp.org/registrar/disco-categories.html>]. It has been used here as a demonstration only. Please refer to the Service Discovery Identities registry document for a list of categories and types and pick the one most suitable for you.

After you have added the two above methods and restarted the server with updated code, have a look at the service discovery window. You should see something like on the screenshot.



Although this was easy, this particular change doesn't affect anything apart from just a visual appearance. Let's get then to more advanced and more useful changes.

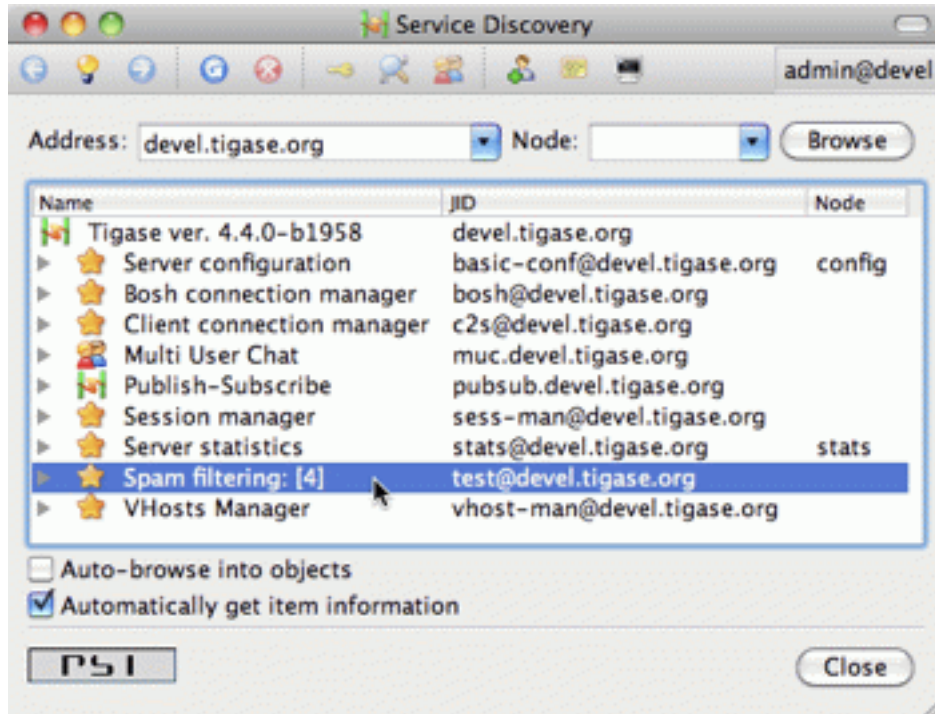
One of the limitations of methods above is that you can not update or change component information at run-time with these methods. They are called only once during `setProperties(...)` method call and the component service discovery information is created and prepared for later use. Sometimes, however it is useful to be able to change the service discovery during run-time.

In our simple spam filtering component let's show how many messages have been checked out as part of the service discovery description string. Every time we receive a message we can to call:

```
updateServiceDiscoveryItem(getName(), null,
    getDiscoDescription() + "-": [" +
        (++messagesCounter) + "-]", true);
```

*A small performance note, in some cases calling `updateServiceDiscoveryItem(...)` might be an expensive operation so probably a better idea would be to call the method not every time we receive a message but maybe every 100 times or so.*

The first parameter is the component JID presented on the service discovery list. However, Tigase server may work for many virtual hosts so the hostname part is added by the lower level functions and we only provide the component name here. The second parameter is the service discovery node which is usually **null** for top level disco elements. Third is the item description (which is actually called *name* in the disco specification). The last parameter specifies if the element is visible to administrators only.



The complete method code is presented below and the screenshot above shows how the element of the service discovery for our component can change if we apply our code and send a few messages to the component.

Using the method we can also add submodes to our component element. The XMPP service discovery really is not for showing application counters, but this case it is good enough to demonstrate the API available in Tigase so we continue with presenting our counters via service discovery. This time, instead of using *null* as a node we put some meaningful texts as in example below:

```
// This is called whenever a message arrives
// to the component
updateServiceDiscoveryItem(getName(), -"messages",
    -"Messages processed: [" + (++messagesCounter) + -"]", true);
// This is called every time the component detects
// spam message
updateServiceDiscoveryItem(getName(), -"spam", -"Spam caught: [" +
    (++totalSpamCounter) + -"]", true);
```

Again, have a look at the full method body below for a complete code example. Now if we send a few messages to the component and some of them are spam (contain words recognized as spam) we can browse the service discovery of the server. Your service discovery should show a list similar to the one presented on the screenshot on the left.

Of course depending on the implementation, initially there might be no sub-nodes under our component element if we call the `updateServiceDiscoveryItem(...)` method only when a message is processed. To make sure that sub-nodes of our component show from the very beginning you can call them in `setProperties(...)` for the first time to populate the service discovery with initial sub-nodes.

Please note, the `updateServiceDiscoveryItem(...)` method is used for adding a new item and updating existing one. There is a separate method though to remove the item:

```
void removeServiceDiscoveryItem(String jid,
```

```
String node, String description)
```

Actually only two first parameters are important: the **jid** and the **node** which must correspond to the existing, previously created service discovery item.

There are two additional variants of the *update* method which give you more control over the service discovery item created. Items can be of different categories and types and can also present a set of features.

The simpler is a variant which sets a set of features for the updated service discovery item. There is a document [<http://xmpp.org/registrar/disco-features.html>] describing existing, registered features. We are creating an example which is going to be a spam filter and there is no predefined feature for spam filtering but for purpose of this guide we can invent two feature identification strings and set it for our component. Let's call update method with following parameters:

```
updateServiceDiscoveryItem(getName(), null, getDiscoDescription(),
    true, -"tigase:x:spam-filter", -"tigase:x:spam-reporting");
```

The best place to call this method is the *setProperties(...)* method so our component gets a proper service discovery settings at startup time. We have set two features for the component disco: *tigase:x:spam-filter* and *tigase:x:spam-reporting*. This method accepts a variable set of arguments so we can pass to it as many features as we need or following Java spec we can just pass an array of **Strings**.

Update your code with call presented above, and restart the server. Have a look at the service discovery for the component now.

The last functionality might be not very useful for our case of the spam filtering component, but it is for many other cases like MUC or PubSub for which it is setting proper category and type for the service discovery item. There is a document listing all currently registered service discovery identities (categories and types). Again there is entry for spam filtering. Let's use the *automation* category and *spam-filter* type and set it for our component:

```
updateServiceDiscoveryItem(getName(), null, getDiscoDescription(),
    -"automation", -"spam-filtering", true,
    -"tigase:x:spam-filter", -"tigase:x:spam-reporting");
```

Of course all these setting can be applied to any service discovery create or update, including sub-nodes. And here is a complete code of the component:

```
import java.util.Arrays;
import java.util.Map;
import java.util.logging.Logger;
import tigase.server.AbstractMessageReceiver;
import tigase.server.Packet;
import tigase.util.JIDUtils;
import tigase.xmpp.StanzaType;

public class TestComponent extends AbstractMessageReceiver {

    private static final Logger log =
        Logger.getLogger(TestComponent.class.getName());

    private static final String BAD_WORDS_KEY = -"bad-words";
    private static final String WHITELIST_KEY = -"white-list";
    private static final String PREPEND_TEXT_KEY = -"log-prepend";
    private static final String SECURE_LOGGING_KEY = -"secure-logging";
    private static final String ABUSE_ADDRESS_KEY = -"abuse-address";
```

```
private static final String NOTIFICATION_FREQ_KEY = "-notification-freq";

private String[] badWords = {"word1", "-word2", "-word3"};
private String[] whiteList = {"admin@localhost"};
private String prependText = "-Spam detected: -";
private String abuseAddress = "-abuse@locahost";
private int notificationFrequency = 10;
private int delayCounter = 0;
private boolean secureLogging = false;
private long spamCounter = 0;
private long totalSpamCounter = 0;
private long messagesCounter = 0;

@Override
public void processPacket(Packet packet) {
    // Is this packet a message?
    if ("message" == packet.getElemName()) {
        updateServiceDiscoveryItem(getName(), "-messages",
            "-Messages processed: [" + (++messagesCounter) + "-]", true);
        String from = JIDUtils.getNodeID(packet.getElemFrom());
        // Is sender on the whitelist?
        if (Arrays.binarySearch(whiteList, from) < 0) {
            // The sender is not on whitelist so let's check the content
            String body = packet.getElemCDATA("/message/body");
            if (body != null && !body.isEmpty()) {
                body = body.toLowerCase();
                for (String word -: badWords) {
                    if (body.contains(word)) {
                        log.finest(prependText + packet.toString(secureLogging));
                        ++spamCounter;
                        updateServiceDiscoveryItem(getName(), "-spam", "-Spam caught: [" +
                            (++totalSpamCounter) + "-]", true);
                        return;
                    }
                }
            }
        }
        // Not a SPAM, return it for further processing
        Packet result = packet.swapElemFromTo();
        addOutPacket(result);
    }

    @Override
    public int processingThreads() {
        return Runtime.getRuntime().availableProcessors();
    }

    @Override
    public int hashCodeForPacket(Packet packet) {
        if (packet.getElemTo() != null) {
            return packet.getElemTo().hashCode();
        }
        // This should not happen, every packet must have a destination
    }
}
```



```

        -// address, but maybe our SPAM checker is used for checking
        -// strange kind of packets too....
        if (packet.getElemFrom() != null) {
            return packet.getElemFrom().hashCode();
        }
        -// If this really happens on your system you should look carefully
        -// at packets arriving to your component and decide a better way
        -// to calculate hashCode
        return 1;
    }

    @Override
    public Map<String, Object> getDefaults(Map<String, Object> params) {
        Map<String, Object> defs = super.getDefaults(params);
        defs.put(BAD_WORDS_KEY, badWords);
        defs.put(WHITELIST_KEY, whiteList);
        defs.put(PREPEND_TEXT_KEY, prependText);
        defs.put(SECURE_LOGGING_KEY, secureLogging);
        defs.put(ABUSE_ADDRESS_KEY, abuseAddress);
        defs.put(NOTIFICATION_FREQ_KEY, notificationFrequency);
        return defs;
    }

    @Override
    public void setProperties(Map<String, Object> props) {
        super.setProps(props);
        badWords = (String[])props.get(BAD_WORDS_KEY);
        whiteList = (String[])props.get(WHITELIST_KEY);
        Arrays.sort(whiteList);
        prependText = (String)props.get(PREPEND_TEXT_KEY);
        secureLogging = (Boolean)props.get(SECURE_LOGGING_KEY);
        abuseAddress = (String)props.get(ABUSE_ADDRESS_KEY);
        notificationFrequency = (Integer)props.get(NOTIFICATION_FREQ_KEY);
        updateServiceDiscoveryItem(getName(), null, getDiscoDescription(),
            -"automation", -"spam-filtering", true,
            -"tigase:x:spam-filter", -"tigase:x:spam-reporting");
    }

    @Override
    public synchronized void everyMinute() {
        super.everyMinute();
        if ((++delayCounter) >= notificationFrequency) {
            addOutPacket(Packet.getMessage(abuseAddress, getComponentId(),
                StanzaType.chat, -"Detected spam messages: -" + spamCounter,
                -"Spam counter", null, newPacketId("spam-")));
            delayCounter = 0;
            spamCounter = 0;
        }
    }

    @Override
    public String getDiscoDescription() {
        return -"Spam filtering";
    }

```

```

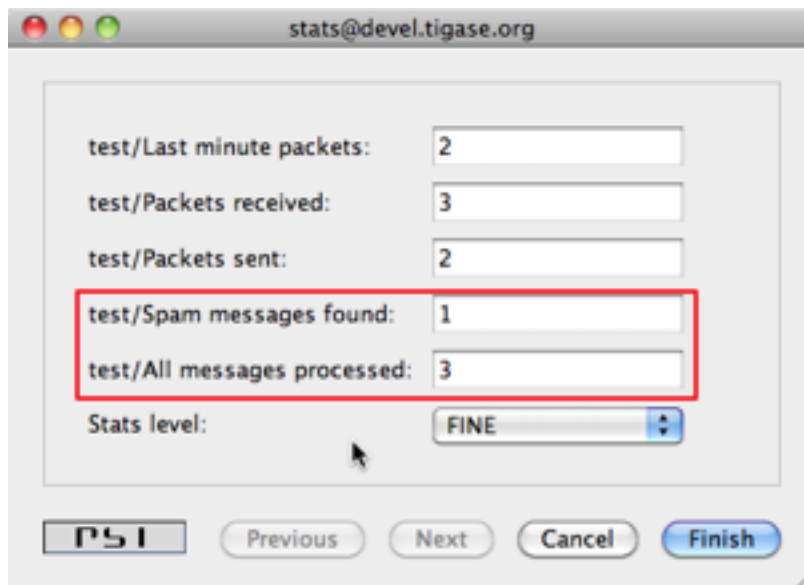
@Override
public String getDiscoCategoryType() {
    return -"spam";
}
}

```

## Component Implementation - Lesson 5 - Statistics

In most cases you'll want to gather some run-time statistics from your component to see how it works, detect possible performance issues or congestion problems. All server statistics are exposed and are accessible via XMPP with ad-hoc commands, HTTP, JMX and some selected statistics are also available via SNMP. As a component developer you don't have to do anything to expose your statistic via any of those protocols, you just have to provide your statistics and the admin will be able to access them any way he wants.

This lesson will teach you how to add your own statistics and how to make sure that the statistics generation doesn't affect application performance.



Your component from the very beginning generates some statistics by classes it inherits. Let's add a few statistics to our spam filtering component:

```

@Override
public void getStatistics(StatisticsList list) {
    super.getStatistics(list);
    list.add(getName(), -"Spam messages found", totalSpamCounter, Level.INFO);
    list.add(getName(), -"All messages processed", messagesCounter, Level.FINER);
    if (list.checkLevel(Level.FINEST)) {
        -// Some very expensive statistics generation code...
    }
}
}

```

I think the code should be pretty much self-explanatory.

You have to call `super.getStatistics(...)` to update stats of the parent class. `StatisticsList` is a collection which keeps all the statistics in a way which is easy to update, search, and retrieve them. You actually don't need to know all the implementation details but if you are interested please refer to the source code and JavaDoc documentation.

The first parameter of the `add(...)` method is the component name. All the statistics are grouped by the component names to make it easier to look at particular component data. Next is a description of the element. The third parameter is the element value which can be any number or string.

The last parameter is probably the most interesting. The idea has been borrowed from the logging framework. Each statistic item has importance level. Levels are exactly the same as for logging methods with **SEVERE** the most critical and **FINEST** the least important. This parameter has been added to improve performance and statistics retrieval. When the `StatisticsList` object is created it gets assigned a level requested by the user. If the `add(...)` method is called with lower priority level then the element is not even added to the list. This saves network bandwidth, improves statistics retrieving speed and is also more clear to present to the end-user.

One thing which may be a bit confusing at first is that, if there is a numerical element added to statistics with **0** value then the Level is always forced to **FINEST**. The assumption is that the administrator is normally not interested **zero-value** statistics, therefore unless he intentionally request the lowest level statistics he won't see elements with **zeros**.

The **if** statement requires some explanation too. Normally adding a new statistics element is not a very expensive operation so passing it with `add(...)` method at an appropriate level is enough. Sometimes, however preparing statistics data may be quite expensive, like reading/counting some records from database. Statistics can be collected quite frequently therefore it doesn't make sense to collect the statistics at all if there not going to be used as the current level is higher then the item we pass anyway. In such a case it is recommended to test whether the element level will be accepted by the collection and if not skip the whole processing altogether.

As you can see, the API for generating and presenting component statistics is very simple and straightforward. Just one method to overwrite and a simple way to pass your own counters. Below is the whole code of the example component:

```
import java.util.Arrays;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;
import tigase.server.AbstractMessageReceiver;
import tigase.server.Packet;
import tigase.stats.StatisticsList;
import tigase.util.JIDUtils;
import tigase.xmpp.StanzaType;

public class TestComponent extends AbstractMessageReceiver {

    private static final Logger log =
        Logger.getLogger(TestComponent.class.getName());

    private static final String BAD_WORDS_KEY = -"bad-words";
    private static final String WHITELIST_KEY = -"white-list";
    private static final String PREPEND_TEXT_KEY = -"log-prepend";
    private static final String SECURE_LOGGING_KEY = -"secure-logging";
```

```

private static final String ABUSE_ADDRESS_KEY = -"abuse-address";
private static final String NOTIFICATION_FREQ_KEY = -"notification-freq";

private String[] badWords = {"word1", -"word2", -"word3"};
private String[] whiteList = {"admin@localhost"};
private String prependText = -"Spam detected: -";
private String abuseAddress = -"abuse@locahost";
private int notificationFrequency = 10;
private int delayCounter = 0;
private boolean secureLogging = false;
private long spamCounter = 0;
private long totalSpamCounter = 0;
private long messagesCounter = 0;

@Override
public void processPacket(Packet packet) {
    -// Is this packet a message?
    if ("message" == packet.getElemName()) {
        updateServiceDiscoveryItem(getName(), -"messages",
            -"Messages processed: [" + (++messagesCounter) + -"]", true);
        String from = JIDUtils.getNodeID(packet.getElemFrom());
        -// Is sender on the whitelist?
        if (Arrays.binarySearch(whiteList, from) < 0) {
            -// The sender is not on whitelist so let's check the content
            Stringbody = packet.getElemCDATA("/message/body");
            if (body != null && !body.isEmpty()) {
                body = body.toLowerCase();
                for (String word -: badWords) {
                    if (body.contains(word)) {
                        log.finest(prependText + packet.toString(secureLogging));
                        ++spamCounter;
                        updateServiceDiscoveryItem(getName(), -"spam", -"Spam caught: [" +
                            (++totalSpamCounter) + -"]", true);
                        return;
                    }
                }
            }
        }
        -// Not a SPAM, return it for further processing
        Packet result = packet.swapElemFromTo();
        addOutPacket(result);
    }
}

@Override
public int processingThreads() {
    return Runtime.getRuntime().availableProcessors();
}

@Override
public int hashCodeForPacket(Packet packet) {
    if (packet.getElemTo() != null) {
        return packet.getElemTo().hashCode();
    }
}

```

```

-// This should not happen, every packet must have a destination
-// address, but maybe our SPAM checker is used for checking
-// strange kind of packets too....
if (packet.getElemFrom() != null) {
    return packet.getElemFrom().hashCode();
}
-// If this really happens on your system you should look carefully
-// at packets arriving to your component and decide a better way
-// to calculate hashCode
return 1;
-}

@Override
public Map<String, Object> getDefaults(Map<String, Object> params) {
    Map<String, Object> defs = super.getDefaults(params);
    defs.put(BAD_WORDS_KEY, badWords);
    defs.put(WHITELIST_KEY, whiteList);
    defs.put(PREPEND_TEXT_KEY, prependText);
    defs.put(SECURE_LOGGING_KEY, secureLogging);
    defs.put(ABUSE_ADDRESS_KEY, abuseAddress);
    defs.put(NOTIFICATION_FREQ_KEY, notificationFrequency);
    return defs;
-}

@Override
public void setProperties(Map<String, Object> props) {
    super.setProperties(props);
    badWords = (String[])props.get(BAD_WORDS_KEY);
    whiteList = (String[])props.get(WHITELIST_KEY);
    Arrays.sort(whiteList);
    prependText = (String)props.get(PREPEND_TEXT_KEY);
    secureLogging = (Boolean)props.get(SECURE_LOGGING_KEY);
    abuseAddress = (String)props.get(ABUSE_ADDRESS_KEY);
    notificationFrequency = (Integer)props.get(NOTIFICATION_FREQ_KEY);
    updateServiceDiscoveryItem(getName(), null, getDiscoDescription(),
        -"automation", -"spam-filtering", true,
        -"tigase:x:spam-filter", -"tigase:x:spam-reporting");
-}

@Override
public synchronized void everyMinute() {
    super.everyMinute();
    if ((++delayCounter) >= notificationFrequency) {
        addOutPacket(Packet.getMessage(abuseAddress, getComponentId(),
            StanzaType.chat, -"Detected spam messages: -" + spamCounter,
            -"Spam counter", null, newPacketId("spam-")));
        delayCounter = 0;
        spamCounter = 0;
    }
-}

@Override
public String getDiscoDescription() {
    return -"Spam filtering";
}

```

```
-}

@Override
public String getDiscoCategoryType() {
    return "spam";
}

@Override
public void getStatistics(StatisticsList list) {
    super.getStatistics(list);
    list.add(getName(), "Spam messages found", totalSpamCounter, Level.INFO);
    list.add(getName(), "All messages processed", messagesCounter, Level.FINE);
    if (list.checkLevel(Level.FINEST)) {
        -// Some very expensive statistics generation code...
    }
}
}
```

## Component Implementation - Lesson 6 - Scripting Support

Scripting support is a basic API built-in to Tigase server and automatically available to any component at no extra resource cost. This framework, however, can only access existing component variables which are inherited by your code from parent classes. It can not access any data or any structures you added in your component. A little effort is needed to expose some of your data to the scripting API.

This guide shows how to extend existing scripting API with your component specific data structures.

Integrating your component implementation with the scripting API is as simple as the code below:

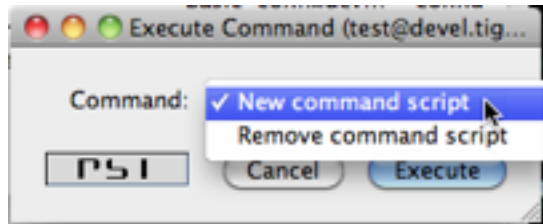
```
private static final String BAD_WORDS_VAR = "badWords";
private static final String WHITE_LIST_VAR = "whiteList";

@Override
public void initBindings(Bindings binds) {
    super.initBindings(binds);
    binds.put(BAD_WORDS_VAR, badWords);
    binds.put(WHITE_LIST_VAR, whiteList);
}
```

This way you expose two the component variables: `badWords` and `whiteList` to scripts under names the same names - two defined constants. You could use different names of course but it is always a good idea to keep things straightforward, hence we use the same variable names in the component and in the script.

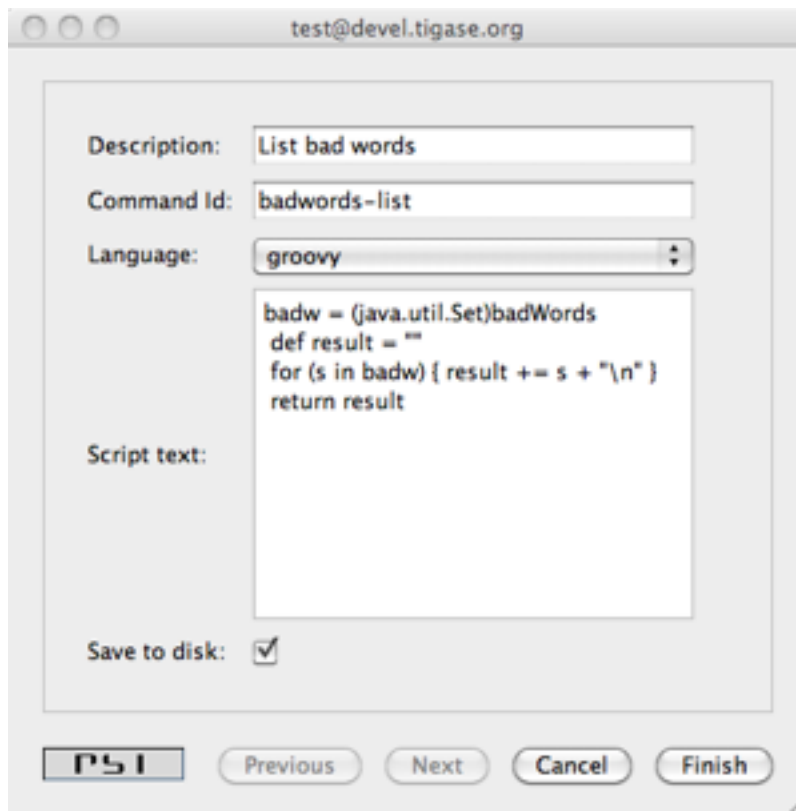
Almost done, almost... In our old implementation these two variables are Java arrays of **String\*s**. **Therefore we can only change their elements but we can not add or remove elements from these structures inside the script. This is not very practical and it puts some serious limits on the script's code. To overcome this problem I have changed the test component code to keep bad words and whitelist in \*java.util.Set collection.** This gives us enough flexibility to manipulate data.

As our component is now ready to cooperate with the scripting API, I will demonstrate now how to add remove or change elements of these collections using a script and ad-hoc commands.



First, browse the server service discovery and double click on the test component. If you use Psi [<http://psi-im.org/>] client this should bring to you a new window with ad-hoc commands list. Other clients may present available ad-hoc commands differently.

The screenshot below shows how this may look. You have to provide some description for the script and an ID string. We use Groovy in this guide but you can as well use any different scripting language.



Please refer to the Tigase scripting documentation for all the details how to add support for more languages. From the Tigase API point of view it all looks the same. You have to select a proper language from the pull-down list on windows shown on the right. If your preferred language is not on the list, it means it is not installed properly and Tigase is unable to detect it.

The script to pull a list of current bad words can be as simple as the following Groovy code:

```
def badw = (java.util.Set)badWords
def result = -""
for (s in badw) { result += s + -"\n" -}
return result
```

As you see from the code, you have to reference your component variables to a variables in your script to make sure a correct type is used. The rest is very simple and is a pure scripting language stuff.

Load the script on to the server and execute it. You should receive a new window with a list of all bad words currently used by the spam filter.

Below is another simple script which allows updating (adding/removing) bad words from the list.

```
import tigase.server.Command
import tigase.server.Packet

def WORDS_LIST_KEY = -"words-list"
def OPERATION_KEY = -"operation"
def REMOVE = -"Remove"
def ADD = -"Add"
def OPERATIONS = [ADD, REMOVE]

def badw = (java.util.Set)badWords
def Packet p = (Packet)packet
def words = Command.getFieldValue(p, WORDS_LIST_KEY)
def operation = Command.getFieldValue(p, OPERATION_KEY)

if (words == null) {
  -// No data to process, let's ask user to provide
  -// a list of words
  def res = (Packet)p.commandResult(Command.DataType.form)
  Command.addFieldValue(res, WORDS_LIST_KEY, "-", -"Bad words list")
  Command.addFieldValue(res, OPERATION_KEY, ADD, -"Operation",
    (String[])OPERATIONS, (String[])OPERATIONS)
  return res
}

def words_list = words.tokenize(",")

if (operation == ADD) {
  words_list.each { badw.add(it.trim()) -}
  return -"Words have been added."
}

if (operation == REMOVE) {
  words_list.each { badw.remove(it.trim()) -}
  return -"Words have been removed."
}

return -"Unknown operation: -" + operation
```

These two scripts are just the beginning. The possibilities are endless and with the simple a few lines of code in your test component you can then extend your application at runtime with scripts doing various things; you can reload scripts, add and remove them, extending and modifying functionality as you need. No need to restart the server, no need to recompile the code and you can use whatever scripting language you like.

Of course, scripts for whitelist modifications would look exactly the same and it doesn't make sense to attach them here.

Here is a complete code of the test component with the new method described at the beginning and data structures changed from array of **String\*s** to **Java \*Set**:

```
import java.util.Arrays;
```



```

import java.util.Collections;
import java.util.Map;
import java.util.Set;
import java.util.concurrent.CopyOnWriteArraySet;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.script.Bindings;
import tigase.server.AbstractMessageReceiver;
import tigase.server.Packet;
import tigase.stats.StatisticsList;
import tigase.util.JIDUtils;
import tigase.xmpp.StanzaType;

public class TestComponent extends AbstractMessageReceiver {

    private static final Logger log =
        Logger.getLogger(TestComponent.class.getName());

    private static final String BAD_WORDS_KEY = -"bad-words";
    private static final String WHITELIST_KEY = -"white-list";
    private static final String PREPEND_TEXT_KEY = -"log-prepend";
    private static final String SECURE_LOGGING_KEY = -"secure-logging";
    private static final String ABUSE_ADDRESS_KEY = -"abuse-address";
    private static final String NOTIFICATION_FREQ_KEY = -"notification-freq";

    private static final String BAD_WORDS_VAR = -"badWords";
    private static final String WHITE_LIST_VAR = -"whiteList";
    private static final String[] INITIAL_BAD_WORDS = {"word1", -"word2", -"word3"};
    private static final String[] INITIAL_WHITE_LIST = {"admin@localhost"};

    /**
     * This might be changed in one threads while it is iterated in
     * processPacket(...) in another thread. We expect that changes are very rare
     * and small, most of operations are just iterations.
     */
    private Set<String> badWords = new CopyOnWriteArraySet<String>();
    /**
     * This might be changed in one threads while it is iterated in
     * processPacket(...) in another thread. We expect that changes are very rare
     * and small, most of operations are just contains(...).
     */
    private Set<String> whiteList = new ConcurrentSkipListSet<String>();
    private String prependText = -"Spam detected: -";
    private String abuseAddress = -"abuse@localhost";
    private int notificationFrequency = 10;
    private int delayCounter = 0;
    private boolean secureLogging = false;
    private long spamCounter = 0;
    private long totalSpamCounter = 0;
    private long messagesCounter = 0;

    @Override
    public void processPacket(Packet packet) {
        // Is this packet a message?

```

```

if ("message" == packet.getElemName()) {
    updateServiceDiscoveryItem(getName(), -"messages",
        -"Messages processed: [" + (++messagesCounter) + -"]", true);
    String from = JIDUtils.getNodeID(packet.getElemFrom());
    -// Is sender on the whitelist?
    if (!whiteList.contains(from)) {
        -// The sender is not on whitelist so let's check the content
        String body = packet.getElemCData("/message/body");
        if (body != null && !body.isEmpty()) {
            body = body.toLowerCase();
            for (String word -: badWords) {
                if (body.contains(word)) {
                    log.finest(prependText + packet.toString(secureLogging));
                    ++spamCounter;
                    updateServiceDiscoveryItem(getName(), -"spam", -"Spam caught: [" +
                        (++totalSpamCounter) + -"]", true);
                    return;
                }
            }
        }
    }
    -// Not a SPAM, return it for further processing
    Packet result = packet.swapElemFromTo();
    addOutPacket(result);
}

@Override
public int processingThreads() {
    return Runtime.getRuntime().availableProcessors();
}

@Override
public int hashCodeForPacket(Packet packet) {
    if (packet.getElemTo() != null) {
        return packet.getElemTo().hashCode();
    }
    -// This should not happen, every packet must have a destination
    -// address, but maybe our SPAM checker is used for checking
    -// strange kind of packets too....
    if (packet.getElemFrom() != null) {
        return packet.getElemFrom().hashCode();
    }
    -// If this really happens on your system you should look carefully
    -// at packets arriving to your component and decide a better way
    -// to calculate hashCode
    return 1;
}

@Override
public Map<String, Object> getDefaults(Map<String, Object> params) {
    Map<String, Object> defs = super.getDefaults(params);
    Collections.addAll(badWords, INITIAL_BAD_WORDS);
    Collections.addAll(whiteList, INITIAL_WHITE_LIST);
}

```

```

        defs.put(BAD_WORDS_KEY, INITIAL_BAD_WORDS);
        defs.put(WHITELIST_KEY, INITIAL_WHITE_LIST);
        defs.put(PREPEND_TEXT_KEY, prependText);
        defs.put(SECURE_LOGGING_KEY, secureLogging);
        defs.put(ABUSE_ADDRESS_KEY, abuseAddress);
        defs.put(NOTIFICATION_FREQ_KEY, notificationFrequency);
        return defs;
    -}

    @Override
    public void setProperties(Map<String, Object> props) {
        super.setProperties(props);
        Collections.addAll(badWords, (String[])props.get(BAD_WORDS_KEY));
        Collections.addAll(whiteList, (String[])props.get(WHITELIST_KEY));
        prependText = (String)props.get(PREPEND_TEXT_KEY);
        secureLogging = (Boolean)props.get(SECURE_LOGGING_KEY);
        abuseAddress = (String)props.get(ABUSE_ADDRESS_KEY);
        notificationFrequency = (Integer)props.get(NOTIFICATION_FREQ_KEY);
        updateServiceDiscoveryItem(getName(), null, getDiscoDescription(),
            -"automation", -"spam-filtering", true,
            -"tigase:x:spam-filter", -"tigase:x:spam-reporting");
    -}

    @Override
    public synchronized void everyMinute() {
        super.everyMinute();
        if ((++delayCounter) >= notificationFrequency) {
            addOutPacket(Packet.getMessage(abuseAddress, getComponentId(),
                StanzaType.chat, -"Detected spam messages: -" + spamCounter,
                -"Spam counter", null, newPacketId("spam-")));
            delayCounter = 0;
            spamCounter = 0;
        }
    -}

    @Override
    public String getDiscoDescription() {
        return -"Spam filtering";
    -}

    @Override
    public String getDiscoCategoryType() {
        return -"spam";
    -}

    @Override
    public void getStatistics(StatisticsList list) {
        super.getStatistics(list);
        list.add(getName(), -"Spam messages found", totalSpamCounter,
            Level.INFO);
        list.add(getName(), -"All messages processed", messagesCounter,
            Level.FINE);
        if (list.checkLevel(Level.FINEST)) {
            -// Some very expensive statistics generation code...

```

```
-}  
-}  
  
@Override  
public void initBindings(Bindings binds) {  
    super.initBindings(binds);  
    binds.put(BAD_WORDS_VAR, badWords);  
    binds.put(WHITE_LIST_VAR, whiteList);  
-}  
  
}
```

## Component Implementation - Lesson 7 - Data Repository

### ConfigRepository

There are cases when you want to store some data permanently by your component. You can of course use the component configuration to provide some database connection settings, implement your own database connector and store records you need. There is, however, a very simple and useful framework which allows you to read and store some data transparently in either a database or a disk file. The framework also supports ad-hoc command interface straight away so you can manipulate your component data using an XMPP client.

In order to use it one needs to extend `tigase.db.comp.ConfigRepository` abstract class.

### RepositoryFactory

In order to have more freedom while accessing repositories it's possible to use `tigase.db.RepositoryFactory` and any of the methods that pertain to desired type of repository one wants to access (auth, user, data):

- `RepositoryFactory.getAuthRepository()`
- `RepositoryFactory.getUserRepository()`
- `RepositoryFactory.getDataRepository()`

Each method takes same set of arguments:

- *class\_name* - qualified name of the class that implements aforementioned repositories type
- *URI* - repository URI
- *params* - map containing additional configuration for the connection.

If there is already available repository for the `<class_name><URI>` identifier then it's returned, otherwise new instance is created.

## Component Implementation - Lesson 8 - Start-up Time

A startup hook in the Tigase is different from the shutdown hook.

This is because you cannot really tell when exactly the startup time is. Is it when the application started, is it when configuration is loaded, is it when all objects are initialized. And this might be even different for each component. Therefore, in fact, there is no startup hook in Tigase in the same sense as the shutdown hook.

There are a few methods which are called at startup time in the following order:

1. **Constructor** - there is of course constructor which has no parameters. However it does not guarantee that this instance of the component will be used at all. The object could be created just to call `getDefaults(...)` and may be destroyed afterwards.
2. **void setName(String name)** - the second call for the component is to set it's unique name within a Tigase instance. It still does not mean too much from the component run-time point of view but some components initialize service discovery data at this point.
3. **void start()** - this is a second void which means the component can start it's internal jobs or worker threads or whatever it needs for future activity. Component's queues and threads are initialized at this point.
4. **Map<String, Object> getDefaults(Map params)** - this is the next call made by configuration manager to collect all the default settings for the component. To help generate default settings, configuration manager passes general properties (starting with `--`) in the Map as parameter to the component. As a result it expects specific settings applicable to the component only (not starting with `--`).
5. **setProperties(Map<String, Object> props)** - after collecting component's defaults, the connection manager combines them with configuration options (not starting with `--`, but starting with the component name) loaded from configuration repository (init.properties file, database, possibly other files). Then the final configuration is passed to the component with `setProperties(...)` method call. Database connections are usually initialized at this point.
6. **void initializationCompleted()** - this method is called for all components after all components are loaded and configuration was set (via `setProperties(...)` method call) for all components.

Therefore, the `initializationCompleted()` hook is the best point if you want to be sure that Tigase server is fully loaded, initialized and functional.

## Configuration API

### Introduction

The component configuration API is quite simple, it consists of two methods:

```
Map<String, Object> getDefaults(Map<String, Object> params);  
void setProperties(Map<String, Object> properties);
```

The first method retrieves configuration defaults from the component while the second sets the new configuration for the component. Although it looks simple, and it is, we should go over some details in order to use them more effectively.

### Component Startup Sequence

Before we go into all the details it might be helpful to know the full component initialization sequence, how the component is brought to life and when the configuration is set. The component loading and starting sequence looks like this:

1. Component class is loaded and a new class instance is created using public constructor with no parameters.
2. Component `setName(compName)`; method is called to set a name for the component. This method is (should) be called only once in the components operation.
3. Component `start()`; method is called which starts all the component internal threads. This method, together with `stop()`; can be called many times to put the component processing on hold or restart processing. Developers should normally not be concerned about these, unless he decided to overwrite these methods.
4. Component `getDefaults()`; method is called to retrieve initial settings for the component. This method is normally called only once during operation.
5. User provided configuration is mixed with the component defaults. Settings which the user has provided overwrite existing defaults, leaving the rest unchanged.
6. Component `setProperties()`; is called to set new configuration for the component. This method can be called many times at any point during the component life time.
7. Component `initializationCompleted()`; method is called to notify the component that the global server initialization has been finished. This method is called only once during the server startup time, after all components have been initialized and configured. This method is mainly used by network connection managers which wait with activating socket listeners until the server is fully functional.

The important thing about all the configuration stuff is that the component does not read/ask/request configuration. **The configuration is pushed to the component by the configuration manager.** The `setProperties()` method can be called at any time and any number of times while the server is running. This design allows for the server reconfiguration during runtime. Developers should be aware of this and properly code the method to allow for the component reconfiguration at runtime.

## Configuration API

Both API methods operate on `Map<String, Object>`, hence, essentially the component configuration is just a list of (key, value) pairs. The Object can any of following:

- String
- Integer
- Long
- Double
- Boolean
- Array of any of above

It is guaranteed that if the component returns a default configuration entry in any of above types, the `setProperties()` method sets the configuration entry in the same exact type. This is quite convenient as you can limit type conversions (numbers parsing for example) in your code.

## getDefaults()

```
Map<String, Object> getDefaults(Map<String, Object> params);
```

*This method is normally called only once, just after the component instance has been created. It is used to get some initial settings from the component and create a default/initial configuration which can be modified by the user. It is recommended that the component returns all possible settings with its default values so they can be presented to the end-user for configuration or diagnostic purposes. No component initialisation can take place here and the developer can not assume that this method is called only once. Every time this method is called it should return only defaults not the settings set with `setProperties()`. The `Map<String, Object>` params provided as a parameter to this method can contain some hints or pre-initial parameters which can affect generating default configuration. This is because configuration for some components may be complex and can have many different presets or optimisations depending on the use case. These presets can be used to generate proper default configuration. If the component implementation extends `AbstractMessageReceiver` then the implementation of the method should always look like this:*

```
@Override
public Map<String, Object> getDefaults(Map<String, Object> params) {
    Map defs = super.getDefaults(params);
    defs.put(CONF_ENTRY_KEY, conf_entry_val);
    return defs;
}
```

## setProperties()

```
void setProperties(Map<String, Object> properties);
```

*This method is called to set configuration for the component. It can be called at any time and many times during the server run-time. The configuration will always contain all entries returned by `getDefaults` method but some of them might be overwritten by user provided settings. If the component implementation extends `AbstractMessageReceiver` then the implementation of the method should always look like this:*

```
@Override
public void setProperties(Map properties) {
    super.setProperties(properties);
    int conf_entry_val = (Integer) properties.get(CONF_ENTRY_KEY);
}
```

## Useful Presets

Normally configuration presets depend on the component implementation and are different for each component. There are a few presets however which are often used commonly by different components:

- `test` If set it means that the server runs in a test mode, which may mean different things for different components. The component may use this parameter to turn testing mode on.
- `admins` If set it provides a list of administrator IDs. These user may have special access permissions for the component. They usually can execute administrator ad-hoc commands.
- `user-db-uri` If set it contains the main database connection string. The component may keep its own data.

## Global Configuration Settings

There are some global settings which are provided to all components and can be used by all of them. Usually they point to some shared resources which can be used by all components.

- **SHARED\_USER\_REPO\_PROP\_KEY** is a configuration key for the user repository instance. This instance can be shared among components and used to store component data in database as well as access to user data. To access the use repository instance you can use the following code:

```
UserRepository user_repo;  
user_repo = (UserRepository) properties.get(RepositoryFactory.SHARED_USER_REPO_PRO
```

- **SHARED\_USER\_REPO\_POOL\_PROP\_KEY** is a configuration key for the user repository pool which in most cases is just an SQL database. To improve the access to the database a connection pool is created which is realized by creating many UserRepository instances connecting to the same database. To access the use repository instance you can use the following code:

```
UserRepository user_repo;  
user_repo = (UserRepository) properties.get(RepositoryFactory.SHARED_USER_REPO_POO
```

- **SHARED\_AUTH\_REPO\_PROP\_KEY** is a configuration key for the authentication repository. Components normally do not need access to this repository unless they deal with user authentication and authentication data is kept separately from the rest of the user data. To access the use repository instance you can use the following code:

```
AuthRepository auth_repo;  
auth_repo = (AuthRepository) properties.get(RepositoryFactory.SHARED_AUTH_REPO_PRO
```

## Packet Filtering in Components

### The Packet Filter API

Tigase server offers an API to filter packet traffic inside every component. You can separately filter incoming and outgoing packets.

By filtering we mean intercepting a packet and possibly making some changes to the packet or just blocking the packet completely. By blocking we mean stopping from any further processing and just dropping the packet.

The packet filtering is based on the `PacketFilterIfc` [<https://projects.tigase.org/projects/tigase-server/repository/changes/src/main/java/tigase/server/PacketFilterIfc.java>] interface. Please have a look in the JavaDoc documentation to this interface for all the details. The main filtering method is `+Packet filter(Packet packet);` which takes packets as an input, processes it, possibly alerting the packet content (may add or remove some payloads) and returns a **Packet** for further processing. If it returns **null** it means the packet is blocked and no further processing is permitted otherwise it returns a **Packet** object which is either the same object it received as a parameter or a modified copy of the original object.

Please note, although **Packet** object is not unmodifiable instance it is recommended to not make any changes on the existing object. The same **Packet** might be processed at the same time by other components or threads, therefore modification of the **Packet** may lead to unpredictable results.

Please refer to an example code in `PacketCounter` [<https://projects.tigase.org/projects/tigase-server/repository/changes/src/main/java/tigase/server/PacketFilterIfc.java>] which is a very simple filter counting different types of packets. This filter is by default loaded to all components which might be very helpful for assessing traffic shapes on newly deployed installation. You can get counters for all types of packets, where they are generated, where they flow, what component they put the most load on.

This is because packet filter can also generate and present its own statistics which are accessible via normal statistics monitoring mechanisms. To take advantage of the statistics functionality the packet filter has



to implement `void getStatistics(StatisticsList list);` method. Normally the method can be empty but you can generate and add to the list own statistics from the filter. Please refer to `PacketCounter` [<https://projects.tigase.org/projects/tigase-server/repository/changes/src/main/java/tigase/server/filters/PacketCounter.java>] for an example implementation code.

## Configuration

Packet filters are configurable, that is a list of packet filters can be provided in Tigase server's configuration for each component separately and for each traffic direction. This gives you a great flexibility and control over the data flow inside the Tigase server.

You can for example, load specific packet filters to all connections managers to block specific traffic or specific packet source from sending messages to users on your server. You could also reduce the server overall load by removing certain payload from all packets. The possibilities are endless.

The default configuration is generated in such a way that each component loads a single packet filter - `PacketCounter` for each traffic direction:

```
message-router/incoming-filters=tigase.server.filters.PacketCounter
message-router/outgoing-filters=tigase.server.filters.PacketCounter
sess-man/incoming-filters=tigase.server.filters.PacketCounter
sess-man/outgoing-filters=tigase.server.filters.PacketCounter
c2s/incoming-filters=tigase.server.filters.PacketCounter
c2s/outgoing-filters=tigase.server.filters.PacketCounter
s2s/incoming-filters=tigase.server.filters.PacketCounter
s2s/outgoing-filters=tigase.server.filters.PacketCounter
bosh/incoming-filters=tigase.server.filters.PacketCounter
bosh/outgoing-filters=tigase.server.filters.PacketCounter
muc/incoming-filters=tigase.server.filters.PacketCounter
muc/outgoing-filters=tigase.server.filters.PacketCounter
```

Now, let's say you have a packet filter implemented in class: **com.company.SpamBlocker**. You want to disable `PacketCounter` on most of the components leaving it only in the message router component and you want to install `SpamBlocker` in all connection managers.

*Please note, in case of the connection managers incoming and outgoing traffic is probably somehow opposite from what you would normally expect.*

- **incoming** is traffic which is submitted to a component by message router and has to be further processed. For connection managers this further processing means sending it out to the network.
- **outgoing** is traffic which is *generated* by the component and goes out of the component. Such a packet is submitted to message router which then decides where to send it for further processing. For connection managers **outgoing** traffic is all the packets just received from the network.

According to that we have to apply the `SpamBlocker` filter to all *outgoing* traffic in all connection managers. You may also decide that it might be actually useful to compare traffic shape between Bosh connections and standard XMPP c2s connections. So let's leave packet counters for this components too.

Here is our new configuration applying `SpamBlocker` to connection managers and `PacketCounter` to a few other components:

```
message-router/incoming-filters=tigase.server.filters.PacketCounter
message-router/outgoing-filters=tigase.server.filters.PacketCounter
sess-man/incoming-filters=
```

```
sess-man/outgoing-filters=  
c2s/incoming-filters=tigase.server.filters.PacketCounter  
c2s/outgoing-filters=tigase.server.filters.PacketCounter,com.company.SpamBlocker  
s2s/incoming-filters=  
s2s/outgoing-filters=com.company.SpamBlocker  
bosh/incoming-filters=tigase.server.filters.PacketCounter  
bosh/outgoing-filters=tigase.server.filters.PacketCounter,com.company.SpamBlocker  
muc/incoming-filters=  
muc/outgoing-filters=
```

The simplest way to apply the new configuration is via the `init.properties` file which is in details described in the *Admin Guide*.

## EventBus API in Tigase

EventBus is a custom publish-subscribe mechanism which allows for the use of Event Listener within Tigase Server. EventBus consists of two separated parts: Distributed EventBus and Local EventBus. Local EventBus is only concerned with local event listener, and will operate events locally. Distributed EventBus is designed to distribute events among cluster nodes. For a more detailed overview of EventBus and its features, please visit The Administration Guide [[http://docs.tigase.org/tigase-server/snapshot/Administration\\_Guide/html/#eventBus](http://docs.tigase.org/tigase-server/snapshot/Administration_Guide/html/#eventBus)].

## EventBus API

To create instance of EventBus use the following code:

```
EventBus eventBus = EventBusFactory.getInstance();
```

**NOTE:** Remember, that EventBus is asynchronous. All handlers are called in a different thread than the thread that initially fired the event.

## Distributed EventBus

Distributed EventBus is designed to distribute events among cluster nodes. Events must extends `tigase.xml.Element`:

```
<EventName xmlns="tigase:demo">  
  <sample_value>1</sample_value>  
</EventName>
```

Events are identified by two elements: name of event and namespace.

## Registering events handlers

To catch and handle an event published in any node of cluster, `EventsHandler` must be registered first.

```
EventHandler handler = new EventHandler() {  
    @Override  
    public void onEvent(String name, String xmlns, Element event) {  
        -// TODO  
    }  
};  
  
eventBus.addHandler("EventName", -"tigase:demo", handler);
```

It is possible to register handler for all events with a specific xmlns such as tigase:demo below:

```
eventBus.addHandler(null, -"tigase:demo", handler);
```

Events created on others cluster node, will have attribute `remote` set to `true` and attribute `source` set to event creator node name:

```
<EventName xmlns="tigase:demo" remote="true" source="node1.example">
  <sample_value>1</sample_value>
</EventName>
```

## Publishing events

The only limitation for events are the requirements of name and xmlns. Internal structure may be defined by programmer.

```
Element event = new Element("EventName", new String[]{"xmlns"}, new String[]{"tigase:demo"});
event.addChild(new Element("sample_value", -"1"));
```

```
eventBus.fire(event);
```

This event will be received by all handlers that are registered for exactly this event, or all events using the **tigase:demo** namespace on all cluster nodes. It is possible to limit event delivery only to the current Tigase instance (current cluster node), by setting the attribute `local`:

```
Element event = new Element("EventName", new String[]{"xmlns", -"local"}, new String[]{"tigase:demo"});
event.addChild(new Element("sample_value", -"1"));
```

```
eventBus.fire(event);
```

## Local EventBus

Local EventBus is the mechanism to distribute events to all listeners on the same instance of Tigase Server. Local EventBus uses Java Objects as events and allows for the transmission instance of object (for example Map or Set).

## Defining events and handlers classes

Local EventBus uses own structures of events and handlers.

### SampleEvent.java.

```
public static class SampleEvent implements Event {

    private final String data;

    public SampleEvent(String data) {
        this.data = data;
    }

    public String getData() {
        return data;
    }
}
```

```
}
```

## Registering events handlers

To catch an event, `EventHandler` must be registered in `EventBus`:

```
EventHandler handler = new EventHandler() {  
    @Override  
    public void onEvent(Event event) {  
  
        -}  
};  
  
eventBus.addHandler(SampleEvent.class, handler);
```

The other way to register a handler is by using annotations. Event consumer class must contain the method with a single parameter, and its type must be equal to expected event type.

**SampleConsumer.java.**

```
public static class SampleConsumer {  
  
    @HandleEvent  
    public void onCatchSomeNiceEvent(SampleEvent event) {  
    }  
  
    @HandleEvent  
    public void onEvent01(ImportantEvent event) {  
    }  
}
```

The instance of class must be registered in `Eventbus`:

```
eventBus.registerAll(consumer);
```

Once this is in place, `EventBus` will be added as the event handler for two different events.

## Publishing events

Publishing events is simple:

```
SampleEvent event = new SampleEvent("data");  
eventBus.fire(event);
```

## Cluster Map Interface

Starting with v7.1.0, a cluster map interface has been implemented. The cluster map is aided by use of the distributed event bus system to communicate between all clusters.

## Requirements

Any full distribution of Tigase will support the Cluster Map API so long as the eventbus component is not disabled. JDK v1.8 is required for this feature, however since Tigase v1.7.0 requires this, you should already have it installed.

The cluster map is stored in memory and follows the `map.util.interface` java standards can be used to improve cluster connections, and help clustered servers keep track of each other.

## Map creation

Map must be created with the following command:

```
java.util.Map<String, String> map = ClusterMapFactory.get().createMap("type",Strin
```

Where "type" is the map ID. This creates the map locally and then fires an event to all clustered servers. Each cluster server has an event handler waiting for, in this case, `NewMapCreate` event. `Map Key` class and `Map Value` class are used to type conversion. Arrays of strings are parameters, for example ID of user session.

Once received, the distributed eventbus will create a local map.

```
eventBus.addHandler(MapCreatedEventHandler.MapCreatedEvent.class, new MapCreatedEv
    @Override
    public void onMapCreated(Map map, String type, String... parameters) {
        -}
    } );
```

A brief example of a map creation is shown here:

```
java.util.Map<String, String> map = ClusterMapFactory.get().createMap("Very_Import
```

This will fire event `MapCreatedEvent` on all other cluster nodes. Strings `"Very_Important_Map_In_User_Session"` and `"user-session-identifier-123"` are given as parameters in `onMapCreated()` method. The event consumer code must know what to do with map with type `"Very_Important_Map_In_User_Session"`. It may retrieve user session `"user-session-identifier-123"` and put this map in this session. It should be used to tell other nodes how to treat the event with a newly created map, and it should be stored in user session.

## Map Changes

Changes to the map on one cluster will trigger `AddValue` or `RemoveValue` events in eventbus. Stanzas sent between clusters will look something like this:

```
<ElementAdd xmlns="tigase:clustered:map">
  <uid>1-2-3</uid>
  <item>
    <key>xKEY</key>
    <value>xVALUE</value>
  </item>
  <item>
    <key>yKEY</key>
    <value>yVALUE</value>
  </item>
</ElementAdd>
```

Code to handle adding an item:

```
eventBus.addHandler(ElementAdd, tigase:clustered:map, new EventHandler() {
    @Override
    public void onEvent(String name, String xmlns, Element event) {
```

```
-});
```

Where the element *event* is the UID, and the name string is the name of the map key/value pair.

This example removes an element from the cluster map. Removal of items look similar:

```
<ElementRemove xmlns="tigase:clustered:map">
  <uid>1-2-3</uid>
  <item>
    <key>xKEY</key>
    <value>xVALUE</value>
  </item>
</ElementRemove>
```

with the code also being similar:

```
eventBus.addHandler(ElementRemove, tigase:clustered:map, new EventHandler() {
    @Override
    public void onEvent(String name, String xmlns, Element name) {
        -});
```

## Map Destruction

Java Garbage Collector will normally remove a local map if it is no longer used. Clustered maps however are not removed in this manner. These maps must be destroyed manually if they are no longer used:

```
ClusterMapFactory.get().destroyMap(cimap);
```

Calling this, the map named *cimap* will be destroyed on each cluster node.

The event handler will catch event when map is destroyed on another cluster node:

```
eventBus.addHandler(MapDestroyedEventHandler.MapDestroyedEvent.class, new MapDestroyedEventHandler() {
    @Override
    public void onMapDestroyed(Map mapX, String type) {
        -}
});
```

---

# Chapter 5. Plugin Development

This is a set of documents explaining details what is a plugin, how they are designed and how they work inside the Tigase server. The last part of the documentation explains step by step creating the code for a new plugin.

- Writing Plugin Code
- Plugin Configuration
- How Packets are Processed by the SM and Plugins
- SASL Custom Mechanisms and Configuration

## Writing Plugin Code

Stanza processing takes place in 4 steps. A different kind of plugin is responsible for each step of processing:

1. `XMPPPreprocessorIfc` [<https://projects.tigase.org/projects/tigase-server/repository/changes/src/main/java/tigase/xmpp/XMPPPreprocessorIfc.java>] - is the interface for packets pre-processing plugins.
2. `XMPPProcessorIfc` [<https://projects.tigase.org/projects/tigase-server/repository/changes/src/main/java/tigase/xmpp/XMPPProcessor.java>] - is the interface for packets processing plugins.
3. `XMPPPostprocessorIfc` [<https://projects.tigase.org/projects/tigase-server/repository/changes/src/main/java/tigase/xmpp/XMPPPostprocessorIfc.java>] - is the interface for packets post-processing plugins.
4. `XMPPPacketFilterIfc` [<https://projects.tigase.org/projects/tigase-server/repository/changes/src/main/java/tigase/xmpp/XMPPPacketFilterIfc.java>] - is the interface for processing results filtering.

If you look inside any of these interfaces you will only find a single method. This is where all the packet processing takes place. All of them take a similar set of parameters and below is a description for all of them:

- **Packet packet** - packet is which being processed. This parameter may never be null. Even though this is not an immutable object it mustn't be altered. None of it's fields or attributes can be changed during processing.
- **XMPPResourceConnection session** - user session which keeps all the user session data and also gives access to the user's data repository. It allows for the storing of information in permanent storage or in memory only during the life of the session. This parameter can be null if there is no online user session at the time of the packet processing.
- **NonAuthUserRepository repo** - this is a user data storage which is normally used when the user session (parameter above) is null. This repository allows for a very restricted access only. It allows for storing some user private data (but doesn't allow overwriting existing data) like messages for offline users and it also allows for reading user public data like VCards.
- **Queue<Packet> results** - this a collection with packets which have been generated as input packet processing results. Regardless a response to a user request is sent or the packet is forwarded to it's destination it is always required that a copy of the input packet is created and stored in the **results** queue.

- **Map<String, Object> settings** - this map keeps plugin specific settings loaded from the Tigase server configuration. In most cases it is unused, however if the plugin needs to access an external database that this is a way to pass the database connection string to the plugin.

After a closer look in some of the interfaces you can see that they extend another interface: **XMP-PImpIfc** [<https://projects.tigase.org/projects/tigase-server/repository/changes/src/main/java/tigase/xmpp/XMPPImpIfc.java>] which provides a basic meta information about the plugin implementation. Please refer to JavaDoc [<http://docs.tigase.org/tigase-server/snapshot/javadoc/tigase/xmpp/impl/package-summary.html>] documentation for all details.

For purpose of this guide we are implementing a simple plugin for handling all **<message/>** packets that is forwarding packets to the destination address. Incoming packets are forwarded to the user connection and outgoing packets are forwarded to the external destination address. This message plugin [<https://projects.tigase.org/projects/tigase-server/repository/changes/src/main/java/tigase/xmpp/impl/Message.java>] is actually implemented already and it is available in our Git repository. The code has some comments inside already but this guide goes deeper into the implementation details.

First of all you have to choose what kind of plugin you want to implement. If this is going to be a packet processor you have to implement the **XMPPProcessorIfc** interface, if this is going to be a pre-processor then you have to implement the **XMPPPreprocessorIfc** interface. Of course your implementation can implement more than one interface, even all of them. There are also two abstract helper classes, one of which you should use as a base for all you plugins **XMPPProcessor** or use **AnnotatedXMPPProcessor** for annotation support.

## Using annotation support

The class declaration should look like this (assuming you are implementing just the packet processor):

```
public class Message extends AnnotatedXMPPProcessor
    implements XMPPProcessorIfc
```

The first thing to create is the plugin **ID**. This is a unique string which you put in the configuration file to tell the server to load and use the plugin. In most cases you can use XMLNS if the plugin wants packets with elements with a very specific name space. Of course there is no guarantee there is no other packet for this specific XML element too. As we want to process all messages and don't want to spend whole day on thinking about a cool ID, let's say our ID is: *message*.

A plugin informs about it's presence using a static **ID** field and **@Id** annotation placed on class:

```
@Id(ID)
public class Message extends AnnotatedXMPPProcessor
    implements XMPPProcessorIfc {
    protected static final String ID = -"message";
}
```

As mentioned before, this plugin receives only this kind of packets for processing which it is interested in. In this example, the plugin is interested only in packets with **<message/>** elements and only if they are in the **"jabber:client"** namespace. To indicate all supported elements and namespaces we have to add 2 more annotations:

```
@Id(ID)
@Handles({
    @Handle(path={ -"message" -}, xmlns="jabber:client")
})
public class Message extends AnnotatedXMPPProcessor
```



```

    implements XMPPProcessorIfc {
    private static final String ID = -"message";
}

```

## Using older non-annotation based implementation

The class declaration should look like this (assuming you are implementing just the packet processor):

```

public class Message extends XMPPProcessor
    implements XMPPProcessorIfc

```

The first thing to create is the plugin **ID** like above.

A plugin informs about it's ID using following code:

```

private static final String ID = -"message";
public String id() { return ID; -}

```

As mentioned before this plugin receives only this kind of packets for processing which it is interested in. In this example, the plugin is interested only in packets with **<message/>** elements and only if they are in **"jabber:client"** namespace. To indicate all supported elements and namespaces we have to add 2 more methods:

```

public String[] supElements() {
    return new String[] {"message"};
}

public String[] supNamespaces() {
    return new String[] {"jabber:client"};
}

```

## Implementation of processing method

Now we have our plugin prepared for loading in Tigase. The next step is the actual packet processing method. For the complete code, please refer to the plugin in the Git. I will only comment here on elements which might be confusing or add a few more lines of code which might be helpful in your case.

```

@Override
public void process(Packet packet, XMPPResourceConnection session,
    NonAuthUserRepository repo, Queue<Packet> results, Map<String, Object> settings)
    throws XMPPException {

    // For performance reasons it is better to do the check
    // before calling logging method.
    if (log.isLoggable(Level.FINEST)) {
        log.log(Level.FINEST, -"Processing packet: {0}", packet);
    }

    // You may want to skip processing completely if the user is offline.
    if (session == null) {
        return;
    }    -// end of if (session == null)

    try {

```

```
// Remember to cut the resource part off before comparing JIDs
BareJID id = (packet.getStanzaTo() != null) ? packet.getStanzaTo().getBareJID() : null;

// Checking if this is a packet TO the owner of the session
if (session.isUserId(id)) {

    // Yes this is message to -'this' client
    Packet result = packet.copyElementOnly();

    // This is where and how we set the address of the component
    // which should receive the result packet for the final delivery
    // to the end-user. In most cases this is a c2s or Bosh component
    // which keep the user connection.
    result.setPacketTo(session.getConnectionId(packet.getStanzaTo()));

    // In most cases this might be skipped, however if there is a
    // problem during packet delivery an error might be sent back
    result.setPacketFrom(packet.getTo());

    // Don't forget to add the packet to the results queue or it
    // will be lost.
    results.offer(result);

    return;
}    -// end of else

// Remember to cut the resource part off before comparing JIDs
id = (packet.getStanzaFrom() != null) ? packet.getStanzaFrom().getBareJID() : null;

// Checking if this is maybe packet FROM the client
if (session.isUserId(id)) {

    // This is a packet FROM this client, the simplest action is
    // to forward it to its destination:
    // Simple clone the XML element and....
    // -... putting it to results queue is enough
    results.offer(packet.copyElementOnly());

    return;
}

// Can we really reach this place here?
// Yes, some packets don't even have from or to address.
// The best example is IQ packet which is usually a request to
// the server for some data. Such packets may not have any addresses
// And they usually require more complex processing
// This is how you check whether this is a packet FROM the user
// who is owner of the session:
JID jid = packet.getFrom();

// This test is in most cases equal to checking getElemFrom()
if (session.getConnectionId().equals(jid)) {
```

```
// Do some packet specific processing here, but we are dealing
// with messages here which normally need just forwarding
Element el_result = packet.getElement().clone();

// If we are here it means FROM address was missing from the
// packet, it is a place to set it here:
el_result.setAttribute("from", session.getJID().toString());

Packet result = Packet.packetInstance(el_result, session.getJID(),
    packet.getStanzaTo());

// -... putting it to results queue is enough
results.offer(result);
}
} catch (NotAuthorizedException e) {
    log.warning("NotAuthorizedException for packet: -" + packet);
    results.offer(Authorization.NOT_AUTHORIZED.getResponseMessage(packet,
        "You must authorize session first.", true));
}    -// end of try-catch
}
```

## Plugin Configuration

Plugin configuration is not very straightforward at the moment but we are going to change it soon.

For now, the best and the simplest way to tell the Tigase server to load or not to load the plugin is via `init.properties` file. The `--sm-plugins` property takes a comma separated list of plugin **IDs** to active at the runtime. Please refer to the documentation for a more complete description.

Obviously you have to know the list of standard plugin IDs to add your to the set. There are 2 ways to find out the list. One is the log file: `logs/tigase-console.log`. If you look inside you can find following output:

```
Loading plugin: jabber:iq:register -...
Loading plugin: jabber:iq:auth -...
Loading plugin: urn:ietf:params:xml:ns:xmpp-sasl -...
Loading plugin: urn:ietf:params:xml:ns:xmpp-bind -...
Loading plugin: urn:ietf:params:xml:ns:xmpp-session -...
Loading plugin: roster-presence -...
Loading plugin: jabber:iq:privacy -...
Loading plugin: jabber:iq:version -...
Loading plugin: http://jabber.org/protocol/stats -...
Loading plugin: starttls -...
Loading plugin: vcard-temp -...
Loading plugin: http://jabber.org/protocol/commands -...
Loading plugin: jabber:iq:private -...
Loading plugin: urn:xmpp:ping -...
```

and this is a list of plugins which are loaded in your installation.

Another way is to look inside the session manager source code which has the default list hardcoded:

```
private static final String[] PLUGINS_FULL_PROP_VAL =
    {"jabber:iq:register", "-jabber:iq:auth", "-urn:ietf:params:xml:ns:xmpp-sasl",
    "-urn:ietf:params:xml:ns:xmpp-bind", "-urn:ietf:params:xml:ns:xmpp-session",
```

```
- "roster-presence", - "jabber:iq:privacy", - "jabber:iq:version",
- "http://jabber.org/protocol/stats", - "starttls", - "msgoffline",
- "vcard-temp", - "http://jabber.org/protocol/commands", - "jabber:iq:private",
- "urn:xmpp:ping", - "basic-filter", - "domain-filter"};
```

In you wish to load a plugin outside these defaults, you have to edit the list and add your plugin IDs as a value to the plugin list property. Let's say our plugin ID is **message** as in our all examples:

```
---sm-plugins=jabber:iq:register,jabber:iq:auth,.....,message
```

Assuming your plugin class is in the classpath it will be loaded and used at the runtime.

There is another part of the plugin configuration though. If you looked at the Writing Plugin Code guide you can remember the **Map settings** processing parameter. This is a map of properties you can set in the configuration file and these setting will be passed to the plugin at the processing time.

Again **init.properties** is the place to put the stuff. These kind of properties start with a string: **sess-man/plugins-conf/**, then you add your **plugin ID** and at the end and follow it with key and value pair for your setting:

```
sess-man/plugins-conf/pluginID/key1=val1
sess-man/plugins-conf/pluginID/key2=val2
sess-man/plugins-conf/pluginID/key3=val3
```

It is possible to provide settings for a few plugins within one configuration string by specifying **multiple pluginIDs** separated with a comma like below:

```
sess-man/plugins-conf/plugin1,plugin2,plugin3/key1=val1
```

This will make key/pair setting available only to listed plugins, in above case plugin1, plugin2 and plugin3.

Last but not least - in case you have **omitted plugin ID**:

```
sess-man/plugins-conf/key1=val1
```

then the configured key-value pair will be a global/common plugin setting available to all loaded plugins.

## How Packets are Processed by the SM and Plugins

For Tigase server plugin development it is important to understand how it all works. There are different kind of plugins responsible for processing packets at different stages of the data flow. Please read the introduction below before proceeding to the actual coding part.

### Introduction

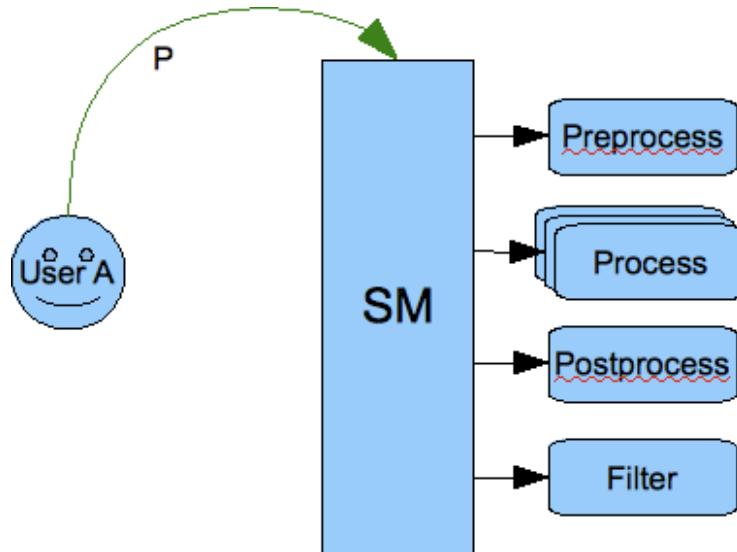
In Tigase server **plugins** are pieces of code responsible for processing particular XMPP stanzas. A separate plugin might be responsible for processing messages, a different one for processing presences, a separate plugins responsible for iq roster, and a different one for iq version and so on.

A plugin provides information about what exact XML element(s) name(s) with xmlns it is interested in. So you can create a plugin which is interested in all packets containing caps child.

There might be no plugin for a particular stanza element, in this case the default action is used which is simple forwarding stanza to a destination address. There might be also more than one plugin for a specific

XML element and then they all process the same stanza simultaneously in separate threads so there is no guarantee on the order in which the stanza is processed by a different plugins.

Each stanza goes through the Session Manager component which processes packets in a few steps. Have a look at the picture below:

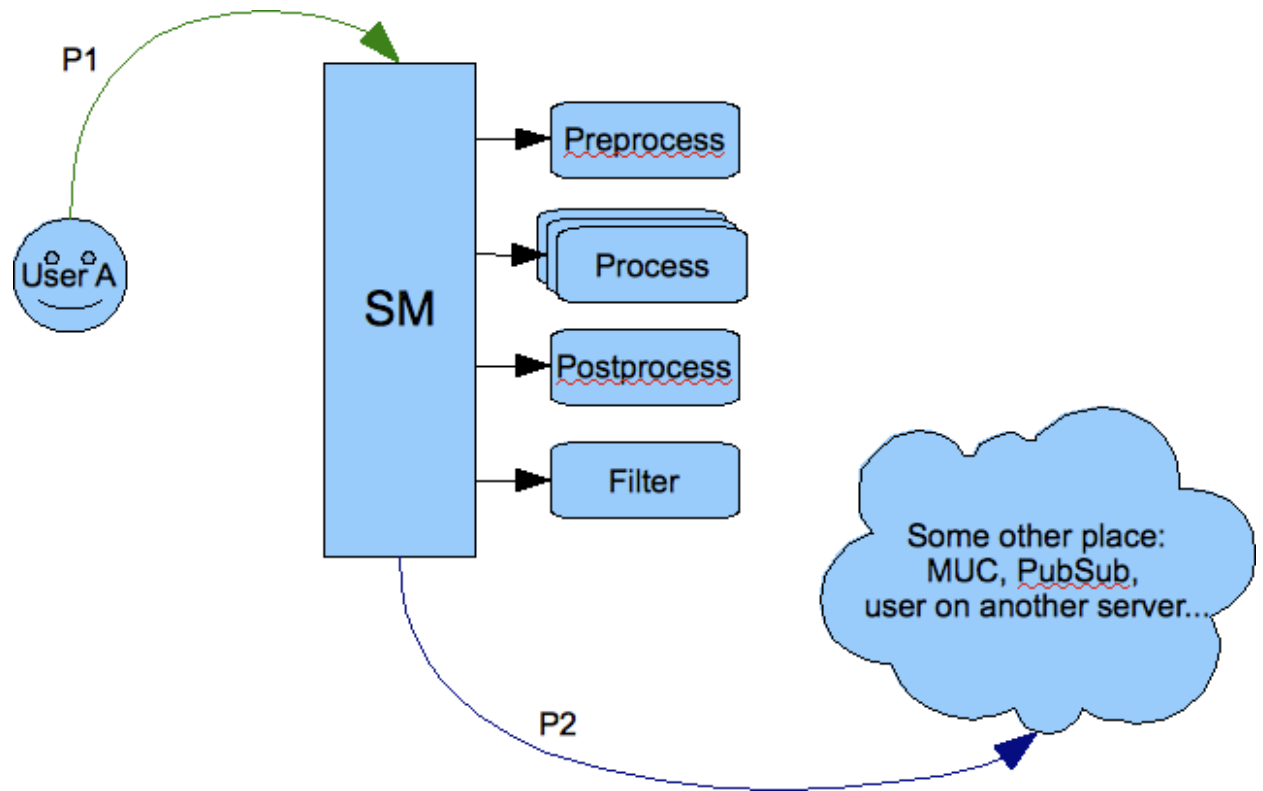


The picture shows that each stanza is processed by the session manager in 4 steps:

1. Pre-processing - All loaded pre-processors receive the packet for processing. They work within session manager thread and they have no internal queue for processing. As they work within Session Manager thread it is important that they limit processing time to absolute minimum as they may affect the Session Manager performance. The intention for the pre-processors is to use them for packet blocking. If the pre-processing result is *true* then the packet is blocked and no further processing is performed.
2. Processing - This is the next step the packet gets through if it wasn't blocked by any of the pre-processors. It gets inserted to all processors queues with requested interest in this particular XML element. Each processor works in a separate thread and has own internal fixed size processing queue.
3. Post-processing - If there is no processor for the stanza then the packet goes through all post-processors. The last post-processor that is built into session manager post-processor tries to apply a default action to a packet which hasn't been processed in step 2. Normally the default action is just forwarding the packet to a destination. Most commonly it is applied to `<message/>` packets.
4. Finally, if any of above 3 steps produced output/result packets all of them go through all filters which may or may not block them.

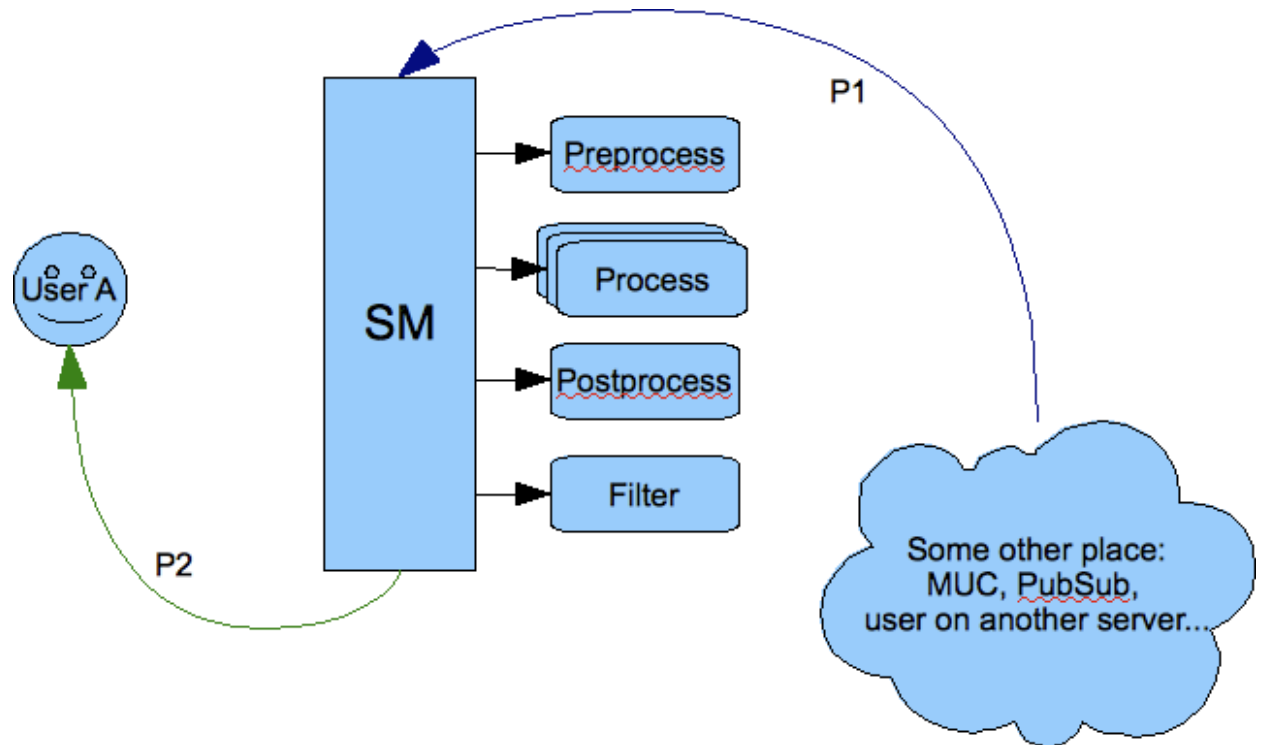
An important thing to note is that we have two kinds or two places where packets may be blocked or filtered out. One place is before packet is processed by the plugin and another place is after processing where filtering is applied to all results generated by the processor plugins.

It is also important to note that session manager and processor plugins act as packet consumers. The packet is taken for processing and once processing is finished the packet is destroyed. Therefore to forward a packet to a destination one of the processor must create a copy of the packet, set all properties and attributes and return it as a processing result. Of course processor can generate any number of packets as a result. Result packets can be generated in any of above 4 steps of the processing. Have a look at the picture below:



If the packet P1 is sent from outside of the server, for example to a user on another server or to some component (MUC, PubSub, transport), then one of the processor must create a copy (P2) of the packet and set all attributes and destination addresses correctly. Packet P1 has been consumed by the session manager during processing and a new packet has been generated by one of the plugins.

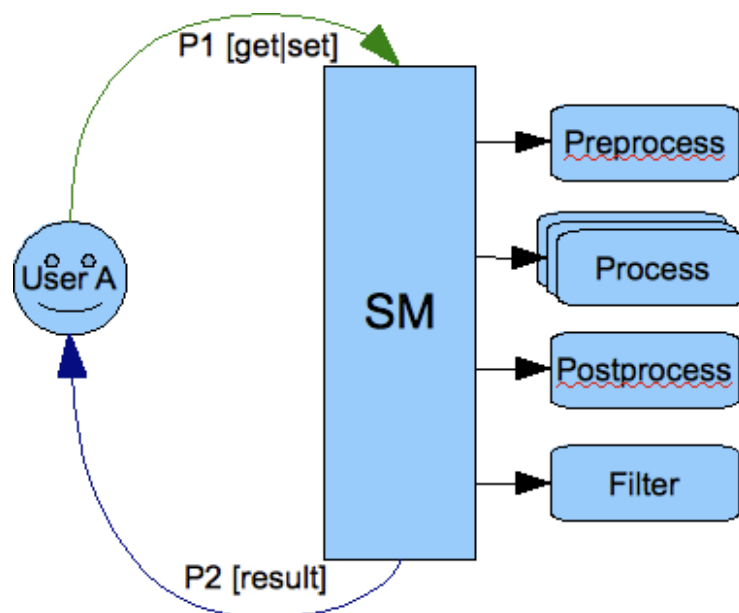
The same of course happens on the way back from the component to the user:



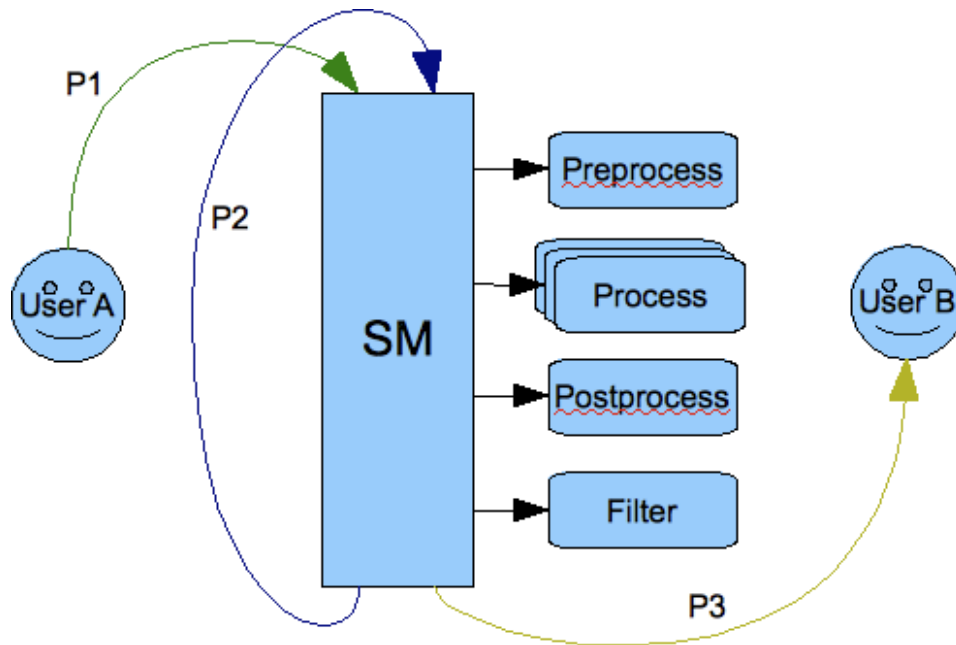
The packet from the component is processed and one of the plugins must generate a copy of the packet to deliver it to the user. Of course packet forwarding is a default action which is applied when there is no plugin for the particular packet.

It is implemented this way because the input packet P1 can be processed by many plugins at the same time therefore the packet should be in fact immutable and must not change once it got to the session manager for processing.

The most obvious processing work flow is when a user sends request to the server and expects a response from the server:



This design has one surprising consequence though. If you look at the picture below showing communication between 2 users you can see that the packet is copied twice before it is delivered to a final destination:



The packet has to be processed twice by the session manager. The first time it is processed on behalf of the User A as an outgoing packet and the second time it is processed on behalf of the User B as an incoming packet.

This is to make sure the User A has permission to send a packet out and all processing is applied to the packet and also to make sure that User B has permission to receive the packet and all processing is applied. If, for example, the User B is offline there is an offline message processor which should send the packet to a database instead of User B.

## SASL Custom Mechanisms and Configuration

This API is available from Tigase XMPP Server version 5.2.0 or later on our current master branch.

*Note that API is under active development. This description may be updated at any time.*

### Basic SASL Configuration

SASL implementation in Tigase XMPP Server is compatible with Java API, the same exact interfaces are used.

The SASL implementation consists of following parts:

1. mechanism
2. CallbackHandler

Properties list for SASL plugin (*sess-man/plugins-conf/urn\:ietf\:params\:xml\:ns\:xmpp-sasl*):

Property	Description
----------	-------------



factory	A factory class for SASL mechanisms. Detailed description at Mechanisms configuration
callbackhandler	A default callback handler class. Detailed description at CallbackHandler configuration
callbackhandler-\${MECHANISM}	A callback handler class for a particular mechanism. Detailed description at CallbackHandler configuration
mechanism-selector	A class for filtering SASL mechanisms available in a stream. Detailed description at Selecting mechanisms

## Mechanisms Configuration

To add a new mechanism, a new factory for the mechanism has to be registered. It can be done with a new line in the `init.properties` file like this one:

```
sess-man/plugins-conf/urn\:ietf\:params\:xml\:ns\:xmpp-sasl/
factory=com.example.OwnFactory
```

The class must implement the `SaslServerFactory` interface. All mechanisms returned by `getMechanismNames()` method will be registered automatically.

The default factory that is available and registered by default is `tigase.auth.TigaseSaslServerFactory` which provides `PLAIN` and `ANONYMOUS` mechanisms.

## CallbackHandler Configuration

The `CallbackHandler` is a helper class used for loading/retrieving authentication data from data repository and providing them to a mechanism.

To register a new callback handler a new line in the `init.properties` file like this one has to be added:

```
sess-man/plugins-conf/urn\:ietf\:params\:xml\:ns\:xmpp-sasl/
callbackhandler=com.example.DefaultCallbackHandler
```

It is also possible to register different callback handlers for different mechanisms:

```
sess-man/plugins-conf/urn\:ietf\:params\:xml\:ns\:xmpp-sasl/
callbackhandler-PLAIN=com.example.PlainCallbackHandler
```

```
sess-man/plugins-conf/urn\:ietf\:params\:xml\:ns\:xmpp-sasl/
callbackhandler-OAUTH=com.example.OAuthCallbackHandler
```

During the authentication process, Tigase server always checks for a handler specific to selected mechanisms, and if there is no specific handler the default one is used.

## Selecting Mechanisms Available in the Stream

The `tigase.auth.MechanismSelector` interface is used for selecting mechanisms available in a stream. Method `filterMechanisms()` should return a collection with mechanisms available based on:

1. all registered SASL factories
2. XMPP session data (from `XMPPResourceConnection` class)

The default selector returns mechanisms from Tigase's default factory (`TigaseSaslServerFactory`) only.

It is possible to use a custom selector by specifying it's class in the `init.properties` file:

```
sess-man/plugins-conf/urn\:ietf\:params\:xml\:ns\:xmpp-sasl/
mechanism-selector=com.example.OwnSelector
```

## Logging/Authentication

After the XMPP stream is opened by a client, the server checks which SASL mechanisms are available for the XMPP session. Depending on whether the stream is encrypted or not, depending on the domain, the server can present different available authentication mechanisms. `MechanismSelector` is responsible for choosing mechanisms. List of allowed mechanisms is stored in the XMPP session object.

When the client/user begins the authentication procedure it uses one particular mechanism. It must use one of the mechanisms provided by the server as available for this session. The server checks whether mechanisms used by the client is on the list of allowed mechanisms. If the check is successful, the server creates `SaslServer` class instance and proceeds with exchanging authentication information. Authentication data is different depending on the mechanism used.

When the SASL authentication is completed without any error, Tigase server should have authorized user name or authorized BareJID. In the first case, the server automatically builds user's JID based on the domain used in the stream opening element in `to` attribute.

If, after a successful authentication, method call: `getNegotiatedProperty("IS_ANONYMOUS")` returns `Boolean.TRUE` then the user session is marked as anonymous. For valid and registered users this can be used for cases when we do not want to load any user data such as roster, vcard, privacy lists and so on. This is a performance and resource usage implication and can be useful for use cases such as support chat. The authorization is performed based on the client database but we do not need to load any XMPP specific data for the user's session.

More details about implementation can be found in the custom mechanisms development section.

## Built-in Mechanisms

**PLAIN**

*TODO!*

**ANONYMOUS**

*TODO!*

## Custom Mechanisms Development

### Mechanism

`getAuthorizationID()` method from `SaslServer` class **should** return bare JID authorized user. In case that the method returns only user name such as **romeo** for example, the server automatically ap-

pend domain name to generate a valid BareJID: *romeo@example.com*. In case the method returns a full, valid BareJID, the server does not change anything.

`handleLogin()` method from `SessionManagerHandler` will be called with user's Bare JID provided by `getAuthorizationID()` (or created later using stream domain name).

## CallbackHandler

For each session authorization, the server creates a new and separate empty handler. Factory which creates handler instance allows to inject different objects to the handler, depending on interfaces implemented by the handler class:

- `AuthRepositoryAware` - injects `AuthRepository`;
- `DomainAware` - injects domain name within which the user attempts to authenticate
- `NonAuthUserRepositoryAware` - injects `NonAuthUserRepository`, although I have no idea what for...

## General Remarks

`JabberIqAuth` used for non-SASL authentication mechanisms uses the same callback as the SASL mechanisms.

Methods `auth` in `Repository` interfaces will be deprecated. These interfaces will be treated as user details providers only. There will be new methods available which will allow for additional login operations on the database such as last successful login recording.

## Known Problems

Because `JabberIqAuth` is initialized separately, we strongly recommend to use more general prefix in **`init.properties`**:

```
sess-man/plugins-conf/${KEY}=${VALUE}
```

instead of

```
sess-man/plugins-conf/urn\:ietf\:params\:xml\:ns\:xmpp-sasl/${KEY}=${VALUE}
```

If `JabberIqAuth` is disabled, then this is not necessary.

---

# Chapter 6. Using Maven

Documents Describing Maven Use with the Tigase Projects

## Setting up Maven in Windows


Here at Tigase, we employ Apache Maven to download latest builds, compile codes for export, and check for errors in the code during build. This guide will go over installing and running Maven from a Windows operating environment. We will consider windows versions 7, 8, and 8.1 for this guide. Because Maven does not come with an installer, there is a manual install process which might be a bit daunting for the new user, but setup and use is fairly simple.

### Requirements

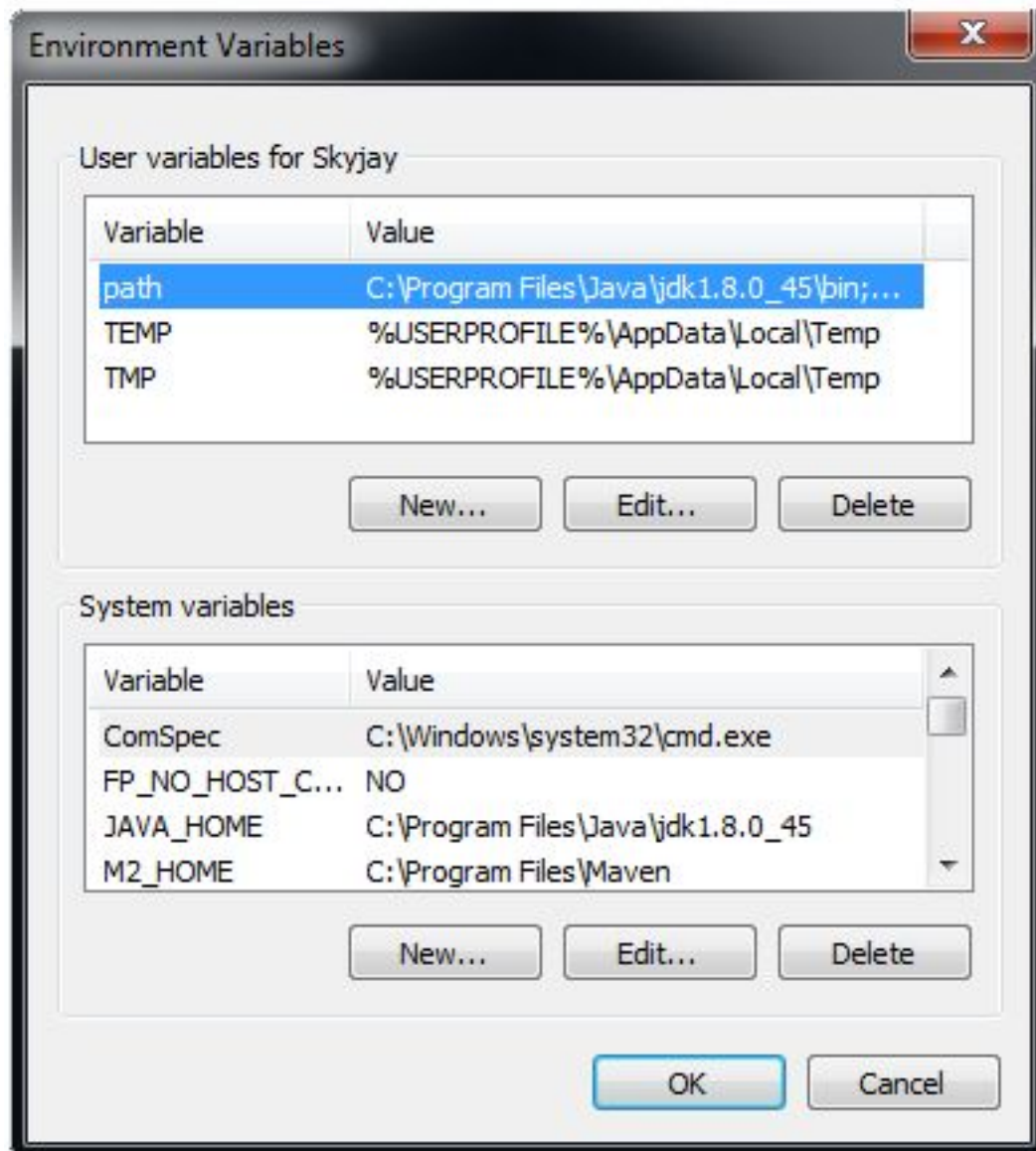
1. Maven requires Java Development Kit (JDK) 6 or later. As Tigase requires the latest JDK to run, that will do for our purposes. If you haven't installed it yet, download the installer from this website [<http://www.oracle.com/technetwork/java/javase/downloads/index.html>]. Once you install JDK and restart your machine, be sure that you have the **JAVA\_HOME** variable entered into Environment Variables so calls to Java will work from the command line.
2. Download the Maven package from here [<https://maven.apache.org/download.cgi>] and unpack it into a directory of your choice. For this guide we will use `C:\Maven\`.

### Setting up Environment Variables

The Environment Variables panel is brought up from the Control Panel by clicking **System and Security**

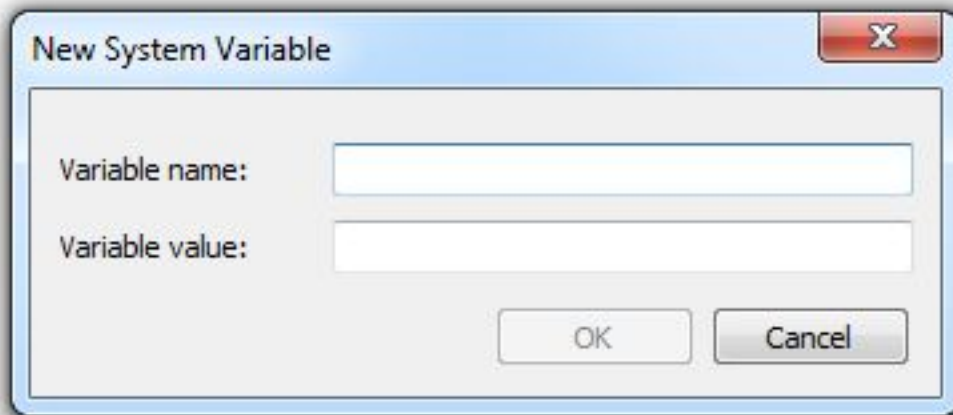
> **System** > **Advanced System Settings**. Now click the  button at the bottom of the panel and the Environment Variables panel will show.

**IMPORTANT NOTICE: CHANGING THESE SETTINGS CAN BREAK OTHER FUNCTIONS IN THE OPERATING SYSTEM. DO NOT FOLLOW THIS GUIDE IF YOU DO NOT KNOW WHAT YOU ARE DOING!**



We need to first add two variable paths to the System variables to account for Maven's install location. As there are some programs that look for M2\_HOME, and others that look for MAVEN\_HOME, it's easier to just add both and have all the bases covered.

Click on New...



For the Name, use M2\_HOME, and for the variable enter the path to maven, which in this case is

C:\Maven

Create another new variable with the MAVEN\_HOME name and add the same directory. **These variable values just point to where you have unpacked maven, so they do not have to be in the C directory.**

Go down to the system variables dialog and select Path, then click on Edit. The Path variables are separated by semicolons, find the end of the Variable value string, and add the following after the last entry:

```
;%M2_HOME%\bin;%MAVEN_HOME%\bin;
```

We have added two variables using the %% wildcards surrounding our Variable names from earlier.

## Testing Maven

Now we must test the command line to be sure everything installed correctly. Bring up the command line either by typing cmd in search, or navigating the start menu.

From the prompt, you do not need to change directory as setting Path allows you to reference it. Type the following command: `mvn -v`

something like this should show up

```
Apache Maven 3.3.3 (7994120775791599e205a5524ec3e0dfe41d4a06; 2015-04-22T04:57:37-07:00)
Maven home: C:\Maven
Java version: 1.8.0_45, vendor: Oracle Corporation
Java home: C:\Program Files\Java\jdk1.8.0_45\jre
Default locale: en_US, platform encoding: Cp1252
OS name: -"windows 7", version: -"6.1", arch: -"amd64", family: -"dos"
```

If you see this message, success! You have finished installation and are ready to use Maven! If not, go back on your settings and insure that JDK is installed, and the JAVA\_HOME, M2\_HOME, and MAVEN\_HOME variables are set properly.

# A Very Short Maven Guide

If you don't use Maven [<http://maven.apache.org/>] at all or use it once a year you may find the document a useful maven commands reminder:

## Snapshot Compilation and Snapshot Package Generation

- `mvn compile` - compilation of the snapshot package
- `mvn package` - create snapshot jar file
- `mvn install` - install in local repository snapshot jar file
- `mvn deploy` - deploy to the remote repository snapshot jar file

## Release Compilation, Generation

- `mvn release:prepare` prepare the project for a new version release
- `mvn release:perform` execute new version release generation

## Generating tar.gz, tar.bz2 File With Sources Only

- `mvn -DdescriptorId=src assembly:assembly`

Any of these commands will work when your commandline is in a directory with a pom.xml file. This file will instruct what Maven will do.

## Maven 2.x Support

Addendum: for a more recent guide please follow Tigase XMPP Server 5.2.0 and Later - Compilation and Generating Distribution Packages.

Thanks to bmalikow [<http://www.tigase.org/user/2>] you can now build Tigase server from sources using Maven 2.x [<http://maven.apache.org/>] tool. This should greatly simplify first steps with Tigase code and it was requested by many of those trying to get the server running from sources. Maven repository with Tigase packages is located at address: maven.tigase.org [<http://maven.tigase.org/>]. Now all you need to compile sources and generate packages needed to run the server is just a few simple steps below:

1. Download and install Maven 2.x
2. Checkout Tigase server sources from Subversion [<http://www.tigase.org/content/>] repository:

```
svn co https://svn.tigase.org/repos/tigase-server/trunk/ tigase-server
```

1. Now go to directory with server code:

```
cd tigase-server
```

1. And run maven command to generate server package:

```
mvn assembly:assembly
```

1. After maven finished his work there should be new subdirectory created: target. Go to this directory now:

```
cd target/
```

1. and list content of this directory. On Linux, Unix system:

```
ls --l
```

+ On MS Windows system:

+

```
dir
```

1. You should see at least 2 files like these:

```
tigase-server-2.4.0-SNAPSHOT-prodenv.tar.gz
tigase-server-2.4.0-SNAPSHOT-prodenv.zip
```

2. Unpack one of these files whichever you like:

```
tar --xzvf tigase-server-2.4.0-SNAPSHOT-prodenv.tar.gz
```

or

```
unzip tigase-server-2.4.0-SNAPSHOT-prodenv.zip
```

3. New directory will be created in our case it will be: tigase-server-2.4.0-SNAPSHOT/. Now go to this directory:

```
cd tigase-server-2.4.0-SNAPSHOT/
```

4. Now almost everything is ready to run the server. Almost because sometimes on Unix like (including Linux) operating systems you have to change script execution bit before you can run it:

```
chmod u+x bin/*
```

5. Now you can run Tigase server:

```
./bin/tigase.sh run etc/tigase.conf
```

You can get a few warnings about missing configuration file (which will be automatically created) and user repository file (which will be automatically created when you register first user).

For your convenience there are a few other startup files in `etc/` directory. You can look and modify them according to your needs.



---

# Chapter 7. Tests

## Tests

Tests are very important part of Tigase server development process.

Each release goes through fully automated testing process. All server functions are considered implemented only when they pass the testing cycle. Tigase test suite is used for all our automatic tests which allows to define different test scenarios.

There is no tweaking on databases for tests. All databases are installed in a standard way and run with default settings. Databases are cleared each time before the test cycle starts.

There are no modifications needed to be made to Tigase's configuration file as well. All tests are performed on a default configuration generated by the configuration wizards.

The server is tested in all supported environments:

1. **XMLDB** - tests with built-in simple XML database. This is a simple and efficient solution for small installations. It is recommended for services with up to 100 user accounts although it has been successfully tested with 10,000 user accounts.
2. **MySQL** - tests with a MySQL [<http://www.mysql.com/>] database. Much slower than XMLDB but may handle many more user accounts.
3. **PostgreSQL** - tests with a PostgreSQL [<http://www.postgresql.org/>] database. Again it is much slower than XMLDB but may handle much more user accounts. This is basically exactly the same code as for MySQL database (SQL Connector) but tests are executed to make sure the code is compatible with all supported SQL databases and to compare performance.
4. **Distributed** - is a test for distributed installation where c2s and s2s components run on separated machine which connects using external an component protocol (XEP-0114 [<http://www.xmpp.org/extensions/xep-0114.html>]) to another machine with SessionManager running.

## Functional Tests

Basic checking to see if all the functions work at correctly. These tests are performed every time the code is sent to source repository.

Version	XMLDB	MySQL	PGSQL	Distributed
3.3.2-b889	00:00:12 [tests/3.3.2-b889/ func/xmldb/function- al-tests.html]	00:00:17 [tests/3.3.2-b889/ func/mysql/function- al-tests.html]	00:00:17 [tests/3.3.2-b889/ func/pgsql/function- al-tests.html]	none
3.3.2-b880	00:00:13 [tests/3.3.2-b880/ func/xmldb/function- al-tests.html]	00:00:15 [tests/3.3.2-b880/ func/mysql/function- al-tests.html]	00:00:15 [tests/3.3.2-b880/ func/pgsql/function- al-tests.html]	None
3.0.2-b700	00:00:22 [tests/3.0.2-b700/ func/xmldb/function- al-tests.html]	00:00:24 [tests/3.0.2-b700/ func/mysql/function- al-tests.html]	00:00:25 [tests/3.0.2-b700/ func/pgsql/function- al-tests.html]	00:00:25 [tests/3.0.2- b700/func/sm- mysql/function- al-tests.html]

2.9.5-b606	00:00:22 [tests/2.9.5-b606/ func/xmlldb/function- al-tests.html]	00:00:24 [tests/2.9.5-b606/ func/mysql/function- al-tests.html]	00:00:24 [tests/2.9.5-b606/ func/pgsql/function- al-tests.html]	00:00:24 [tests/2.9.5- b606/func/sm- mysql/function- al-tests.html]
2.9.3-b548	00:00:22 [tests/2.9.3-b548/ func/xmlldb/function- al-tests.html]	00:00:23 [tests/2.9.3-b548/ func/mysql/function- al-tests.html]	00:00:25 [tests/2.9.3-b548/ func/pgsql/function- al-tests.html]	00:00:25 [tests/2.9.3- b548/func/sm- mysql/function- al-tests.html]
2.9.1-b528	00:00:21 [tests/2.9.1-b528/ func/xmlldb/function- al-tests.html]	00:00:23 [tests/2.9.1-b528/ func/mysql/function- al-tests.html]	00:00:24 [tests/2.9.1-b528/ func/pgsql/function- al-tests.html]	00:00:25 [tests/2.9.1- b528/func/sm- mysql/function- al-tests.html]
2.8.6-b434	00:00:21 [tests/2.8.6-b434/ func/xmlldb/function- al-tests.html]	00:00:24 [tests/2.8.6-b434/ func/mysql/function- al-tests.html]	00:00:24 [tests/2.8.6-b434/ func/pgsql/function- al-tests.html]	00:00:25 [tests/2.8.6- b434/func/sm- mysql/function- al-tests.html]
2.8.5-b422	00:00:21 [tests/2.8.5-b422/ func/xmlldb/function- al-tests.html]	00:00:24 [tests/2.8.5-b422/ func/mysql/function- al-tests.html]	00:00:24 [tests/2.8.5-b422/ func/pgsql/function- al-tests.html]	00:00:26 [tests/2.8.5- b422/func/sm- mysql/function- al-tests.html]
2.8.3-b409	00:00:27 [tests/2.8.3-b409/ func/xmlldb/function- al-tests.html]	00:00:29 [tests/2.8.3-b409/ func/mysql/function- al-tests.html]	00:00:29 [tests/2.8.3-b409/ func/pgsql/function- al-tests.html]	00:00:32 [tests/2.8.3- b409/func/sm- mysql/function- al-tests.html]
2.7.2-b378	00:00:30 [tests/2.7.2-b378/ func/xmlldb/function- al-tests.html]	00:00:34 [tests/2.7.2-b378/ func/mysql/function- al-tests.html]	00:00:33 [tests/2.7.2-b378/ func/pgsql/function- al-tests.html]	00:00:35 [tests/2.7.2- b378/func/sm- mysql/function- al-tests.html]
2.6.4-b300	00:00:30 [tests/2.6.4-b300/ func/xmlldb/function- al-tests.html]	00:00:32 [tests/2.6.4-b300/ func/mysql/function- al-tests.html]	00:00:35 [tests/2.6.4-b300/ func/pgsql/function- al-tests.html]	00:00:39 [tests/2.6.4- b300/func/sm- mysql/function- al-tests.html]
2.6.4-b295	00:00:29 [tests/2.6.4-b295/ func/xmlldb/function- al-tests.html]	00:00:32 [tests/2.6.4-b295/ func/mysql/function- al-tests.html]	00:00:45 [tests/2.6.4-b295/ func/pgsql/function- al-tests.html]	00:00:36 [tests/2.6.4- b295/func/sm- mysql/function- al-tests.html]
2.6.0-b287	00:00:31 [tests/2.6.0-b287/ func/xmlldb/function- al-tests.html]	00:00:34 [tests/2.6.0-b287/ func/mysql/function- al-tests.html]	00:00:47 [tests/2.6.0-b287/ func/pgsql/function- al-tests.html]	00:00:43 [tests/2.6.0- b287/func/sm- mysql/function- al-tests.html]

2.5.0-b279	00:00:30 [tests/2.5.0-b279/ func/xmlldb/function- al-tests.html]	00:00:34 [tests/2.5.0-b279/ func/mysql/function- al-tests.html]	00:00:45 [tests/2.5.0-b279/ func/pgsql/function- al-tests.html]	00:00:43 [tests/2.5.0- b279/func/sm- mysql/function- al-tests.html]
2.4.0-b263	00:00:29 [tests/2.4.0-b263/ func/xmlldb/function- al-tests.html]	00:00:33 [tests/2.4.0-b263/ func/mysql/function- al-tests.html]	00:00:45 [tests/2.4.0-b263/ func/pgsql/function- al-tests.html]	00:00:44 [tests/2.4.0- b263/func/sm- mysql/function- al-tests.html]
2.3.4-b226	None	00:00:48 [tests/ function- al-tests.html]	None	None

## Performance Tests

Checking to see whether the function performs well enough.

Version	XMLDB	MySQL	PGSQL	Distributed
3.3.2-b889	00:12:17 [tests/3.3.2- b889/perf/ xmlldb/perfor- mance-tests.html]	00:13:42 [tests/3.3.2- b889/perf/ mysql/perfor- mance-tests.html]	00:17:10 [tests/3.3.2- b889/perf/ pgsql/perfor- mance-tests.html]	none
3.3.2-b880	00:13:39 [tests/3.3.2- b880/perf/ xmlldb/perfor- mance-tests.html]	00:14:09 [tests/3.3.2- b880/perf/ mysql/perfor- mance-tests.html]	00:17:39 [tests/3.3.2- b880/perf/ pgsql/perfor- mance-tests.html]	None
3.0.2-b700	00:10:26 [tests/3.0.2- b700/perf/ xmlldb/perfor- mance-tests.html]	00:11:00 [tests/3.0.2- b700/perf/ mysql/perfor- mance-tests.html]	00:12:08 [tests/3.0.2- b700/perf/ pgsql/perfor- mance-tests.html]	00:24:05 [tests/3.0.2- b700/perf/sm- mysql/perfor- mance-tests.html]
2.9.5-b606	00:09:54 [tests/2.9.5- b606/perf/ xmlldb/perfor- mance-tests.html]	00:11:18 [tests/2.9.5- b606/perf/ mysql/perfor- mance-tests.html]	00:37:08 [tests/2.9.5- b606/perf/ pgsql/perfor- mance-tests.html]	00:16:20 [tests/2.9.5- b606/perf/sm- mysql/perfor- mance-tests.html]
2.9.3-b548	00:10:00 [tests/2.9.3- b548/perf/ xmlldb/perfor- mance-tests.html]	00:11:29 [tests/2.9.3- b548/perf/ mysql/perfor- mance-tests.html]	00:36:43 [tests/2.9.3- b548/perf/ pgsql/perfor- mance-tests.html]	00:16:47 [tests/2.9.3- b548/perf/sm- mysql/perfor- mance-tests.html]
2.9.1-b528	00:09:46 [tests/2.9.1- b528/perf/ xmlldb/perfor- mance-tests.html]	00:11:15 [tests/2.9.1- b528/perf/ mysql/perfor- mance-tests.html]	00:36:12 [tests/2.9.1- b528/perf/ pgsql/perfor- mance-tests.html]	00:16:36 [tests/2.9.1- b528/perf/sm- mysql/perfor- mance-tests.html]

2.8.6-b434	00:10:02 [tests/2.8.6-b434/perf/xmldb/performance-tests.html]	00:11:45 [tests/2.8.6-b434/perf/mysql/performance-tests.html]	00:36:36 [tests/2.8.6-b434/perf/pgsql/performance-tests.html]	00:17:36 [tests/2.8.6-b434/perf/sm-mysql/performance-tests.html]
2.8.5-b422	00:12:37 [tests/2.8.5-b422/perf/xmldb/performance-tests.html]	00:14:40 [tests/2.8.5-b422/perf/mysql/performance-tests.html]	00:38:59 [tests/2.8.5-b422/perf/pgsql/performance-tests.html]	00:21:40 [tests/2.8.5-b422/perf/sm-mysql/performance-tests.html]
2.8.3-b409	00:12:32 [tests/2.8.3-b409/perf/xmldb/performance-tests.html]	00:14:26 [tests/2.8.3-b409/perf/mysql/performance-tests.html]	00:37:57 [tests/2.8.3-b409/perf/pgsql/performance-tests.html]	00:21:26 [tests/2.8.3-b409/perf/sm-mysql/performance-tests.html]
2.7.2-b378	00:12:28 [tests/2.7.2-b378/perf/xmldb/performance-tests.html]	00:14:57 [tests/2.7.2-b378/perf/mysql/performance-tests.html]	00:37:09 [tests/2.7.2-b378/perf/pgsql/performance-tests.html]	00:22:20 [tests/2.7.2-b378/perf/sm-mysql/performance-tests.html]
2.6.4-b300	00:12:46 [tests/2.6.4-b300/perf/xmldb/performance-tests.html]	00:14:59 [tests/2.6.4-b300/perf/mysql/performance-tests.html]	00:36:56 [tests/2.6.4-b300/perf/pgsql/performance-tests.html]	00:35:00 [tests/2.6.4-b300/perf/sm-mysql/performance-tests.html]
2.6.4-b295	00:12:23 [tests/2.6.4-b295/perf/xmldb/performance-tests.html]	00:14:59 [tests/2.6.4-b295/perf/mysql/performance-tests.html]	00:42:24 [tests/2.6.4-b295/perf/pgsql/performance-tests.html]	00:30:18 [tests/2.6.4-b295/perf/sm-mysql/performance-tests.html]
2.6.0-b287	00:13:50 [tests/2.6.0-b287/perf/xmldb/performance-tests.html]	00:16:53 [tests/2.6.0-b287/perf/mysql/performance-tests.html]	00:48:17 [tests/2.6.0-b287/perf/pgsql/performance-tests.html]	00:49:06 [tests/2.6.0-b287/perf/sm-mysql/performance-tests.html]
2.5.0-b279	00:13:29 [tests/2.5.0-b279/perf/xmldb/performance-tests.html]	00:16:58 [tests/2.5.0-b279/perf/mysql/performance-tests.html]	00:47:15 [tests/2.5.0-b279/perf/pgsql/performance-tests.html]	00:41:52 [tests/2.5.0-b279/perf/sm-mysql/performance-tests.html]
2.4.0-b263	00:13:20 [tests/2.4.0-b263/perf/xmldb/performance-tests.html]	00:16:21 [tests/2.4.0-b263/perf/mysql/performance-tests.html]	00:43:56 [tests/2.4.0-b263/perf/pgsql/performance-tests.html]	00:42:08 [tests/2.4.0-b263/perf/sm-mysql/performance-tests.html]
2.3.4-b226	None	01:23:30 [tests/performance-tests.html]	None	None

## Stability Tests

Checking to see whether the function behaves well in long term run. It must handle hundreds of requests a second in a several hour server run.

Version	XMLDB	MySQL	PGSQL	Distributed
2.3.4-b226	None	16:06:31 [tests/stability-tests.html]	None	None

## Tigase Test Suite

Tigase Test Suite is an engine which allows you to run tests. Essentially it just executes **TestCase** implementations. The tests may depend on other tests which means they are executed in specific order. For example authentication test is executed after the stream open test which in turn is executed after network socket connection test.

Each **TestCase** implementation may have it's own set of specific parameters. There is a set of common parameters which may be applied to any **TestCase**. As an example of the common parameter you can take **-loop = 10** which specified that the **TestCase** must be executed 10 times. The test specific parameter might be **-user-name = tester** which may set the user name for authentication test.

The engine is very generic and allows you to write any kind of tests but for the Tigase projects the current **TestCase** implementations mimic an XMPP client and are designed to test XMPP servers.

The suite contains a kind of scripting language which allows you to combine test cases into a test scenarios. The test scenario may contain full set of functional tests for example, another test scenario may contain performance tests and so on.

## Running Tigase Test Suite (TTS)

To obtain TTS, you will first need to clone the repository

```
git clone https://repository.tigase.org/git/tigase-testsuite.git
```

Once cloning is finished, navigate to the TTS root directory and compile with maven:

```
mvn clean install
```

Maven will compile TTS and place jars in the necessary locations. From the same directory, you can begin running TTS using the following command:

```
./scripts/all-tests-runner.sh
```

You should see the following, which outlines the possible options to customize your test run

```
Run selected or all tests for Tigase server
```

```
----
```

```
Author: Artur Hefczyc <artur_hefczyc@vnu.co.uk>
```

```
Version: 2.0.0
```

```
----
```

```
---help|-h This help message
```

```
---func [mysql|pgsql|derby|mssql|mongodb]
```

```
Run all functional tests for a single database configuration
```

```

---lmem [mysql|pgsql|derby|mssql|mongodb]
    Run low memory tests for a single database configuration
---perf [mysql|pgsql|derby|mssql|mongodb]
    Run all performance tests for a single database configuration
---stab [mysql|pgsql|derby|mssql|mongodb]
    Run all stability tests for a single database
    configuration
---func-all Run all functional tests for all database
    configurations
---lmem-all Run low memory tests for all database
    configurations
---perf-all Run all performance tests for all database
    configurations
---stab-all Run all stability tests for all database
    configurations
---all-tests Run all functionality and performance tests for
    database configurations
---single test_file.cot
---other script_file.xmpt
-----
Special parameters only at the beginning of the parameters list
---debug|-d                Turns on debug mode
---skip-db-relad|-no-db    Turns off reloading database
---skip-server|-no-serv    Turns off Tigase server start
---small-mem|-sm          Run in small memory mode
-----
Other possible parameters are in following order:
[server-dir] [server-ip]

```

## Customizing Tigase Test Suite

You may run the tests from a command line like above, however you may create and edit the /scripts/tests-runner-settings.sh file to fit your Tigase installation and avoid having to have long complex commands as this template shows:

```

#!/bin/bash

func_rep="func-rep.html"
perf_rep="perf-rep.html"
db_name="tigasetest"
db_user="tigase"
db_pass="tigase"
root_user="root"
root_pass="root"

TESTS=("derby" -"mysql" -"pgsql" -"mssql")
IPS=("127.0.0.1" -"127.0.0.1" -"127.0.0.1" -"127.0.0.1")

server_timeout=10

server_dir="/home/tigase/tigase-server"
database="derby"
#database="mysql"
server_ip="127.0.0.1"

```

```
MS_MEM=100
MX_MEM=1000
```

```
SMALL_MS_MEM=10
SMALL_MX_MEM=50
```

This will allow you to maintain identical settings through multiple runs of TTS. See the next section for learning how the scripting language works and how you can create and run your own custom tests.

## Test Suite Scripting Language

The test suite contains scripting language which allows you to combine test cases into a test scenarios. On the lowest level, however the language is designed to allow you to describe the test by setting test parameters, test comments, identification and so on.

Let's look at the example test description.

```
Short name@test-id-1;test-id-2: Short description for the test case
{
  --loop = 10
  --user-name = Frank
  # This is a comment which is ignored
}
>> Long, detailed description of the test case <<
```

Meaning of all elements:

1. **Short name** is any descriptive name you want. It doesn't need to be unique, just something which tells you what this test is about. @ is a separator between the short name and the test ids.
2. **test-id-1;test-id-2** is a semicolon separated of the test cases IDs. The tests cases are executed in the listed order. And listing them there means that the test-id-2 depends on test-id-1. Normally you don't have to list all the dependencies because all mandatory dependencies are included automatically. Which means if you have an authentication test case the suite adds the network socket connection and stream opening tests automatically. Sometimes however, there are dependencies which are optional or multiple mandatory dependencies and you need to select which one has to be executed. As a good example is the authentications test case. There are many authentication tests: PLAIN-AUTH, SASL-DIGESTMD5, SASL-PLAIN, DIGEST-AUTH and they are all mandatory for most of other tests like roster, presence and so on. One of the authentication tests is a default dependency but if you put on the list different authentication it will be used instead of default one.
3. **:** is a separator between test cases ids list and the short test description.
4. **Short test description** is placed between : - colon and opening { - curly bracket. This is usually quite brief, single line test description.
5. **{ }** curly brackets contain all the test parameters, like how many times the test has to be executed or run the test in a separate thread, user name, host IP address for the network connection and many others.
6. **>> <<** inside the double greater than and double less than you put a very long, multiple line test description.

As for the testing script between open curly brackets { and close one } you can put all the test case parameters you wish. The format for it is:

**-parameter-name = value**

Parameter names always start with -. Note, some parameters don't require any value. They can exist on their own without any value assigned:

**-debug-on-error**

This imitates if you were to put **yes** or **true** as the value.

The scripting language includes also support for variables which can be assigned any value and used multiple times later on. You assign a value to the variable the same way as you assign it to the parameter:

**\$(variable-name) = value**

The variable name must be always enclosed with brackets () and start with \$.

The value may be enclosed within double quotes "" or double quotes may be omitted. If this is a simple string like a number or character string consisting only of digits, letters, underscore \_ and hyphen - then you can omit double quotes otherwise you must enclose the value.

The test case descriptions can be nested inside other test case descriptions. Nested test case descriptions inherit parameters and variables from outer test case description.

## Writing Tests for Plugins

You can write tests in a simple text file which is loaded during test suite runtime.

You simply specify what should be send to the server and what response should be expected from the server. No need to write Java code and recompile the whole test suite for new tests. It means new test cases can be now written easily and quickly which hopefully means more detailed tests for the server.

How it works:

Let's take XEP-0049 [<http://www.xmpp.org/extensions/xep-0049.html>] Private XML Storage. Looking into the spec we can see the first example:

Example 1. Client Stores Private Data

**CLIENT:**

```
<iq type="set" id="1001">
  <query xmlns="jabber:iq:private">
    <exodus xmlns="exodus:prefs">
      <defaultnick>Hamlet</defaultnick>
    </exodus>
  </query>
</iq>
```

**SERVER:**

```
<iq type="result" id="1001"/>
```

This is enough for the first simple test. I have to create text file `JabberIqPrivate.test` looking like this:

```
send: {
```



```
<iq type="set" id="1001">
  <query xmlns="jabber:iq:private">
    <exodus xmlns="exodus:prefs">
      <defaultnick>Hamlet</defaultnick>
    </exodus>
  </query>
</iq>
}
```

```
expect: {
<iq type="result" id="1001"/>
}
```

And now I can execute the test:

```
testsuite $ ./scripts/all-tests-runner.sh ---single JabberIqPrivate.test
```

```
Tigase server home directory: -../server
Version: 2.8.5-b422
Database:          xmldb
Server IP:         127.0.0.1
Extra parameters: JabberIqPrivate.test
Starting Tigase:
Tigase running pid=6751
```

```
Running: 2.8.5-b422-xmldb test, IP 127.0.0.1...
Script name: scripts/single-xmpp-test.xmpt
Common test: Common test -... failure!
FAILURE, (Received result doesnt match expected result.,
Expected one of: [<iq id="1001" type="result"/>],
received:
[<iq id="1001" type="error">
  <query xmlns="jabber:iq:private">
    <exodus xmlns="exodus:prefs">
      <defaultnick>Hamlet</defaultnick>
    </exodus>
  </query>
  <error type="cancel">
    <feature-not-implemented xmlns="urn:ietf:params:xml:ns:xmpp-stanzas"/>
    <text xml:lang="en" xmlns="urn:ietf:params:xml:ns:xmpp-stanzas">
      Feature not supported yet.</text>
    </error>
  </iq>]],
```

```
Total: 100ms
Test time: 00:00:02
Shutting down Tigase: 6751
```

If I just started working on this XEP and there is no code on the server side, the result is perfectly expected although maybe this is not what we want. After a while of working on the server code I can execute the test once again:

```
testsuite $ ./scripts/all-tests-runner.sh ---single JabberIqPrivate.test
```

```
Tigase server home directory: -../server

Version: 2.8.5-b422

Database:          xmldb

Server IP:         127.0.0.1

Extra parameters: JabberIqPrivate.test

Starting Tigase:

Tigase running pid=6984

Running: 2.8.5-b422-xmldb test, IP 127.0.0.1...

Script name: scripts/single-xmpp-test.xmpt

Common test:  Common test  -... success,  Total: 40ms

Test time: 00:00:01

Shutting down Tigase: 6984
```

This is it. The result we want in a simple and efficient way. We can repeat it as many times we want which is especially important in longer term trials. Every time we change the server code we can re-run tests to make sure we get correct responses from the server.

You can have a look in the current build, with more complete test cases, file for JabberIqPrivate [<https://projects.tigase.org/projects/tigase-testsuite/repository/revisions/master/entry/tests/data/JabberIqPrivate.cot>].

Now my server tests are no longer outdated. Of course not all cases are so simple. Some XEPs require calculations to be done before stanza is sent or to compare received results. A good example for this case is user authentication like SASL and even NON-SASL. But still, there are many cases which can be covered by simple tests: roster management, privacy lists management, vCard, private data storage and so on.

## Test Case Parameters Description

There is long list of parameters which can be applied to any test case. Here is the description of all possible parameters which can be used to build test scenarios.

## Test Report Configuration

There are test report parameters which must be set in the main script file in order to generate HTML report from the test. These parameters have no effect if they are set inside the test case description.

1. **-version = 2.0.0** sets the test script version. This is used to easily detect incompatibility issues when the test suite loads a script created for more recent version of the suite and may not work properly for this version.
2. **-output-format = (html | html-content)** sets the output format for the test report. There is actually only one format possible right now - HTML. The only difference between these 2 options is that the **html** format creates full HTML page with HTML header and body. The **html-content** format on the

other hand creates only what is inside `<body/>` element. And is used to embed test result inside other HTML content.

3. **-output-file = "report-file.html"** sets the file name for the test report.
4. **-output-history = (yes | no)** sets logging of the all protocol data sent between test suite and the XMPP server. Normally for functional tests it is recommended to set it to **yes** but for all other tests like performance or load tests it should be set to **no**.
5. **-history-format = separate-file** sets protocol data logging to a separate file. Currently this is the only possible option.
6. **-output-cols = (5 | 7)** Only valid values are:
 

```
5: -"Test name", -"Result", -"Test time", -"Description" [, -"History" -]
7: -"Test name", -"Result", -"Total time", -"OK", -"Average", -"Description" [,
```
7. **-title = "The title of the report page"** This parameter sets the test report title which is placed in the HTML page in the `<title/>` element as well as in the first page header.

## Basic Test Parameters

These parameters can be set on per-test case basis but usually they are set in the main script file to apply them to all test cases.

1. **-base-ns = "jabber:client"** sets the XML name space used for the XML stream in the XMPP connection. Some test cases can be used to test client to server protocol as well as server to server protocol and possibly different protocols added in the future.
2. **-debug** switches debugging mode on. All the communication between the test suite and the server is printed out to the text console and all other debugging information including java exceptions are displayed as well. It is especially useful when some test fails and you want to find out why.
3. **-debug-on-error** switches on debugging mode on error detection. Normally debug output generates lots of message which makes the output very hard to read. Especially in the performance tests not only you can read fast scrolling lines of the protocol data but also it slows the test down. This option however turns debugging off if everything is working well and then generates debug output if any test error is detected.
4. **-def-auth = (auth-plain | auth-digest | auth-sasl)** sets the default authentication method for the user connection.
5. **-def-stream = (stream-client | stream-server | stream-component | stream-bosh)** sets the connection stream to be tested and the name space for the connection.
6. **-host = "host.name"** the vhost name the tested server runs for. It may be the real DNS name or just configured for testing purposes hostname. It must match however the server configuration.
7. **-keys-file = "certs/keystore"** sets the location of the keys store file. No need to touch it.
8. **-keys-file-password = keystore** sets the password for the keystore file. Normally you don't have to touch it.
9. **-serverip = "127.0.0.1"** defines the XMPP server IP address. You may omit this parameter and then the IP address will be determined automatically based on the server DNS address. However if the DNS

address can not be correctly resolved or if you run tests on the localhost you can use this parameter to enforce the IP address.

10. **-socket-wait = 10000** sets the network socket timeout in milliseconds that is maximum time the test suite will wait for the response from the server. You may want to increase the timeout for some specific tests which require lots of computation or database activity on the server. Normally 10 seconds is enough for most cases.
11. **-stop-on-fail = true** causes the script to terminate all actions on the first failed test case. It helps diagnosing the server state at the failure point.
12. **-trust-file = "certs/client\_truststore"** sets the file name for the client trust store file. No need to change it.
13. **-trust-file-password = truststore** sets the password for the trust store file. Normally you don't have to touch it.
14. **-user-name = tester** sets the user name used for the XMPP connections between the test suite and the XMPP server. It is usually set globally the same for all tests and for some tests like receiving the server configuration you may want to use a different account (with admin permissions). Then you can set a different user for this specific test case.
15. **-user-pass = tester-password** sets the password for the user used for the XMPP connection between the test suite and the XMPP server.
16. **-user-resr = resource** sets the user JID resource part for the XMPP connection between the test suite and the XMPP server.

## Test Case Parameters

Test parameters which are normally set on per-test case basis and apply only to the test they are set for and all inherited tests. Some of the parameters though are applied only to inherited test cases. Please look in the description below to find more details.

1. **-active-connection** is a similar parameter to **-on-one-socket** option. If set the suite doesn't close the network socket and if the test is run in loop each loop run re-uses the network connection. Unlike in the **-on-one-socket** mode the whole test is executed on each run including XMPP stream initialization and user authentication. This option is currently not recommended in a normal use. It is useful only to debug the server behavior in very special use cases.
2. **-background** executes the test in a separate thread in background and immediately returns control to the test suite program without waiting for the test to complete. Default behavior is to execute all tests sequentially and run next test when previous one has been completed. This parameter however allows to run tests concurrently. This a bit similar option to the **-daemon** parameter. The daemon test/task however is ignored completely and results from the daemon are not collected where the background test is a normal test which is run concurrently with another one or possibly many other tests.
3. **-daemon** creates a task running in background in a separate thread. Such a test runs infinitely as a daemon, it is not recorded in the test report and it's result is not calculated. The purpose of such test/task is to work as a helper for other test cases. A good example of such daemon test is message responder - the test which runs under a different user name and waits for messages and responding to the sender.
4. **-delay = 1000** sets the waiting time in milliseconds after the test case is completed. You may use it if you want to introduce short delay between each test cases run in the loop or if you start the helper daemon thread and you have to add the delay to make sure it is ready to work before next real test starts sending requests to the daemon.

5. **-expect-type = error** sets the type for a packet expected as a response. Some test cases like message sender expects sometimes response with the same type it has sent the packet ( **chat** ) but in some other cases when it sends a message to a user who has privacy lists set to block messages the response should be with an error. This way we can use the same test cases for testing different responses scenarios.
6. **-loop = 10** sets the number of times the test (and all inherited tests) are repeated. You can use a **\$(loop)** pseudo-variable to obtain and use the current loop run number. This is useful if you want to run every loop run for a different user name like registering 10 different user accounts. To do this you stick the **\$(loop)** variable to the user name string: **-user-name = "nick\_name\_\$(loop)"**.
7. **-loop-delay = 10** sets a delay in milliseconds between each individual loop run for the tests which is run multiple times. This is similar parameter to the **-delay** one but the **-delay** option introduces a delay after the whole test (or all loop runs) has been completed. The loop delay options adds waiting time between each run of the looped test.
8. **-loop-start = 5** sets the loop starting value. It doesn't affect number of loop runs in any way. It only affects the value of the **\$(loop)** variable. Let's say you want to run a load test for the server with 100k concurrent users and you want to run the test from 3 different machines. To make sure each machine uses distinct user accounts you have to set a different **-loop-start** parameter on each to prevent from overlapping.
9. **-messages = 10** sets the number of messages to send to the server. This is another way of looping the test. Instead of repeating the whole test with opening network connection, XMPP stream, authentication and so on it causes only to send the message this many times. This parameter is accepted by some test cases only which send messages. For the messages listeners - test cases which is supposed to respond to the messages the number set here specifies how many times the response must be sent before the test successfully terminates its work.
10. **-multi-thread** option causes to run the test case and all inherited in all levels test cases in separate threads. Normally the test case where you put the parameter doesn't have a test ID (what you put between @ and : characters so it doesn't run a test on its own. Instead it contains a series of test cases inside which are then run in a separate thread each. This is a key parameter to run tests for many concurrent users. (Not a load tests though.) For example you can see whether the server behaves correctly when 5 simultaneously modifies their roster. The execution time all inherited tests run in a separate threads is added together and also results from each individual test is calculated and added to the total main test results.
11. **-no-record** is used for kind of configuration tests (tasks) which are used to prepare the XMPP server or database for later tests. As an example can be creation of the test user account which is later on used for the roster tests. Usually you don't want to include such tests in the test report and using this parameter you essentially exclude the test from the report. The test and the result however shows in the command line output so you can still track what is really going on.
12. **-on-one-socket** is a modifier for a looped test case. Normally when we switch looping on using **-loop** parameter the suite resets the state, closes the network socket and runs the test from the very beginning including opening network socket, XMPP stream, authentication and so on. This parameter however changes this behavior. The network socket is not closed when the test run is completed (successfully) and next run executes only the last part of the test omitting the XMPP stream initialization, authentication and all others but last. This is useful when you want to send many messages to the server (although this effect may be accomplished using **-messages** parameter as well) or registering many user accounts on the server, unregistering user accounts and any other which might make sense repeating many times.
13. **-port = 5223** this parameter is similar to the IP address setting and can be also set globally for all tests. Normally however you set it for a selected tests only to check SSL connection. For all other tests default port number is used. Therefore this parameter has been included in this section instead of "Basic test parameters".

14. **-presence** this parameter enables sending initial presence with positive priority after connection and binding the session.
15. **-repeat-script = 100** and **-repeat-wait = 10** are 2 parameters are specific to the common test cases. (The test cases which reads the test input/output data from the pseudo-xml text file. The first parameter is another variation of test looping. It sets how many times the test has to be repeated. It works very much like the **-on-one-socket** parameter. The only difference is that the common test can preserve some internal states between runs and therefore it has more control over the data. The second parameter sets the timeout in milliseconds to wait/delay between each individual test run and it is a very similar parameter to the **-delay** one but it sets a timeout inside the common test instead.
16. **-source-file = "dir/path/to/file.cot"** is a parameter to set the "common test" script file. The common test is a test cases which depends on the authentication test case and can read data to send and responses to expect from the text file. The "cot" file is a pseudo-xml file with stanzas to send and stanzas to expect. The the test cases compares the received packets with those in the text file and reports the test result. This is usually a more convenient way to write a new test cases than coding them in Java.
17. **-time-out-ok** is set for a test case when we expect socket timeout as a correct result from the test case. Normally the timeout means that the test failed and there was no response from the server at all or the response was incorrect. For some tests however (like sending a message to the user who is blocking messages through privacy lists) the timeout is the desired correct test result.
18. **-to-jid = "user\_name@host.name [mailto:user\_name@host.name]"** sets the destination address for packets sending packets somewhere. As an example is the test case sending <message /> packet. You can set the destination address for the packet. Mind, normally every test expects some response for the data sent so make sure the destination end-point will send back the data expected by the test case.

---

# Chapter 8. Experimental

The guide contains description of non-standard or experimental functionality of the server. Some of them are based on never published extensions, some of them are just test implementation for new ideas or performance improvements.

- Dynamic Rosters
- Mobile Optimizations
- Bosh Session Cache

## Dynamic Rosters

### Problem Description

Normal roster contacts stored and created as **dynamic roster parts** are delivered to the end user transparently. The XMPP client doesn't really know what contacts come from its own **static** roster created manually by the user and what contacts come from a **dynamic** roster part; contacts and groups generated dynamically by the server logic.

Some specialized clients need to store extra bits of information about roster contacts. For the normal user **static** roster information can be stored as private data and is available only to this single user. In some cases however, clients need to store information about contacts from the dynamic roster part and this information must be available to all users accessing **dynamic** roster part.

The protocol defined here allows the exchange of information, saving and retrieving extra data about the contacts.

### Syntax and Semantics

Extra contact data is accessed using IQ stanzas, specifically by means of a child element qualified by the **jabber:iq:roster-dynamic** namespace. The child element MAY contain one or more children, each describing a unique contact item. Content of the element is not specified and is implementation dependent. From Tigase's point of view it can contain any valid XML data. Whole element is passed to the DynamicRoster implementation class as is and without any verification. Upon retrieving the contact extra data the DynamicRoster implementation is supposed to provide a valid XML element with all the required data for requested **jid**.

The **jid** attribute specifies the Jabber Identifier (JID) that uniquely identifies the roster item. Inclusion of the **jid** attribute is **REQUIRED**.

Following actions on the extra contact data are allowed:

- **set** - stores extra information about the contact
- **get** - retrieves extra information about the contact

### Retrieving Contact Data

Upon connecting to the server and becoming an active resource, a client can request the extra contact data. This request can be made either before or after requesting the user roster. The client's request for the extra contact data is **OPTIONAL**.

Example: Client requests contact extra data from the server using **get** request:

```
<iq type='get' id='rce_1'>
<query xmlns='jabber:iq:roster-dynamic'>
<item jid='archimedes@eureka.com' />
</query>
</iq>
```

Example: Client receives contact extra data from the server, but there was either no extra information for the user, or the user was not found in the dynamic roster:

```
<iq type='result' id='rce_1'>
<query xmlns='jabber:iq:roster-dynamic'>
<item jid='archimedes@eureka.com' />
</query>
</iq>
```

Example: Client receives contact extra data from the server, and there was some extra information found about the contact:

```
<iq type='result' id='rce_1'>
<query xmlns='jabber:iq:roster-dynamic'>
<item jid='archimedes@eureka.com'>
<phone>+12 3234 322342</phone>
<note>This is short note about the contact</note>
<fax>+98 2343 3453453</fax>
</item>
</query>
</iq>
```

## Updating/Saving Extra Information About the Contact

At any time, a client **MAY** update extra contact information on the server.

Example: Client sends contact extra information using **set** request.

```
<iq type='set' id='a78b4q6ha463'>
<query xmlns='jabber:iq:roster-dynamic'>
<item jid='archimedes@eureka.com'>
<phone>+22 3344 556677</phone>
<note>he is a smart guy, he knows whether the crown is made from pure gold or not.
</item>
</query>
</iq>
```

Client responds to the server:

```
<iq type='result' id='a78b4q6ha463' />
```

A client **MAY** update contact extra information for more than a single item in one request:

Example: Client sends contact extra information using **set** request with many `<item/>` elements.

```
<iq type='set' id='a78b4q6ha464'>
<query xmlns='jabber:iq:roster-dynamic'>
```



```
<item jid='archimedes@eureka.com'>
<phone>+22 3344 556677</phone>
<note>he is a smart guy, he knows whether the crown is made from pure gold or not.
</item>
<item jid='newton@eureka.com'>
<phone>+22 3344 556688</phone>
<note>He knows how heavy I am.</note>
</item>
<item jid='pascal@eureka.com'>
<phone>+22 3344 556699</phone>
<note>This guy helped me cure my sickness!</note>
</item>
</query>
</iq>
```

Client responds to the server:

```
<iq type='result' id='a78b4q6ha464' />
```

## Configuration

DynamicRoster implementation class should be configured in the **init.properties** file. As it's an extension to the **PresenceState**, **PresenceSubscription** and **Roster** plugins classes should be configured either for each plugin:

```
sess-man/plugins-conf/jabber\:iq\:roster/dynamic-roster-classes=<class list>
sess-man/plugins-conf/presence-state/dynamic-roster-classes=<classes list>
sess-man/plugins-conf/presence-subscription/dynamic-roster-classes=<classes list>
```

or globally:

```
sess-man/plugins-conf/dynamic-roster-classes=<classes list>
```

<classes list> is a comma separated list of classes.

## Mobile Optimizations

### Problem Description

In default configuration stanzas are sent to the client when processing is finished, but in mobile environment sending or receiving data drains battery due to use of the radio.

To save energy data should be sent to client only if it is important or client is waiting for it.

### Solution

When mobile client is entering inactive state it notifies server about it by sending following stanza:

```
<iq type="set" id="xx">
<mobile
  xmlns="http://tigase.org/protocol/mobile#v3"
  enable="true"/>
</iq>
```

After receiving stanza server starts queuing stanza which should be send to mobile client. What kind of queued stanzas depends on the plugins used and in case of **Mobile v3** presence stanzas are queued as well as message stanzas which are Message Carbons. Any other stanza (such as iq or plain messenge) is sent immediately to the client and every stanza from queue is also sent at this time.

When mobile client is entering active state it notifies server by sending following stanza:

```
<iq type="set" id="xx">
<mobile
  xmlns="http://tigase.org/protocol/mobile#v3"
  enable="false"/>
</iq>
```

After receiving stanza server sends all queued stanzas to the client.

Also all stanzas from queue will be sent if number of stanzas in queue will reach queue size limit. By default this limit is set to 50.

## Queuing Algorithms

There are three mobile optimization plugins for Tigase:

- **Mobile v1** - all presence stanzas are kept in queue
- **Mobile v2** - only last presence from each source is kept in queue
- **Mobile v3** - only last presence from each source is kept in queue, also Message Carbons are queued

If you wish to activate you Mobile v1 plugin you need to send presented above with xmlns attribute value replaced with *http://tigase.org/protocol/mobile#v1*

If you wish to activate you Mobile v2 plugin you need to send presented above with xmlns attribute value replaced with *http://tigase.org/protocol/mobile#v2*

## Configuration

Mentioned plugins are not activated by default thus additional entries in the init.properties are required:

```
++sm-plugins=+mobile_v1,+mobile_v2,+mobile_v3
```

**Only one of these plugins should be enabled**

## Bosh Session Cache

### Problem Description

Web clients have no way to store any data locally, on the client side. Therefore after a web page reload the web clients loses all the context it was running in before the page reload.

Some elements of the context can be retrieved from the server like the roster and all contacts presence information. Some other data however, can not be restored easily like opened chat windows and the chat windows contents. Even if the roster restoring is possible, this operation is very expensive in terms of time and resources on the server side.

One of possible solutions is to allow web client to store some data in the Bosh component cache on the server side for the time while the Bosh session is active. After the page reloads, if the client can somehow retrieve SID (stored in cookie or provided by the web application running the web client) it is possible to reload all the data stored in the Bosh cache to the client.

Bosh session context data are: roster, contacts presence information, opened chat windows, chat windows content and some other minor data. Ideally the web client should be able to store any data in the Bosh component cache it wants.

## Bosh Session Cache Description

The Bosh Session Cache is divided into 2 parts - automatic cache and dynamic cache.

The reason for splitting the cache into 2 parts is that some data can be collected automatically by the Bosh component and it would be very inefficient to require the client to store the data in the Bosh cache. The best example for such data is the Roster and contacts presence information.

- **automatic cache** - is the cache part which is created automatically by the Bosh component without any interaction with the client. The client, however, can access the cache at any time. I would say this is a read-only cache but I don't want to stop client from manipulating the cache if it needs. The client usually, only retrieves data from this part of the cache as all changes should be automatically updated by the Bosh component. The general idea for the automatic cache is that the data stored there are accessible in the standard XMPP form. So no extra code is needed for processing them.
- **dynamic cache** - is the cache part which is or can be modified at any time by the client. Client can store, retrieve, delete and modify data in this part of the cache.

## Cache Protocol

All the Bosh Session Cache actions are executed using additional `<body/>` element attributes: `cache` and `cache-id`. Attribute `cache` stores the action performed on the Bosh cache and the `cache-id` attribute refers to the `cache` element if the action attribute needs it. `cache-id` is optional. There is a default cache ID (empty one) associated with the elements for which the `cache-id` is not provided.

If the `<body/>` element contains the `cache` attribute it means that all data included in the `<body/>` refer to the cache action. It is not allowed, for example to send a message in the body and have the cache action set to **get**. The `<body/>` element with cache action **get**, **get\_all**, **on**, **off**, **remove** must be empty. The `<body/>` element with actions **set** or **add** must contain data to store in the cache.

## Cache Actions

- **on** or **off** - the client can switch the cache on or off at any time during the session. It is recommended, however that the client switches the cache **on** in the first body packet, otherwise some information from the automatic cache may be missing. The automatic cache is created from the stream of data passing the Bosh component. Therefore if the cache is switched on after the roster retrieval is completed then the roster information will be missing in the cache. If the cache is set to **off** (the default value) all requests to the cache are ignored. This is to ensure backward compatibility with the original Bosh specification and to make sure that in a default environment the Bosh component doesn't consume any extra resources for cache processing and storing as the cache wouldn't be used by the client anyway.
- **get** - retrieves the cache element pointing by the `cache-id` from the Bosh cache. Note there is no **result** cache action. The `<body/>` sent as a response from the server to the client may contain cache results for a given `cache-id` and it may also contain other data received by the Bosh component for the client. It may also happen that large cached data are split into a few parts and each part can be sent in a separate `<body/>` element. It may usually happen for the Roster data.

- **get\_all** - retrieves all the elements kept in the Bosh cache. That action can be performed after the page reload. The client doesn't have to request every single cached item one by one. It can retrieve all cache items in one go. It doesn't mean however the whole cache is sent to the client in a single `<body/>` element. The cache content will be divided into a smaller parts of a reasonable size and will be sent to the client in a separate `<body/>` elements. It may also happen that the `<body/>` element contain the cache elements as well as the new requests sent to the user like new messages or presence information.
- **set** - sends data to the Bosh Session cache for later retrieval. The client can store any data it wants in the cache. The Bosh components stores in the cache under the selected ID all the data inside the `<body/>` element. The only restriction is that the cached data must be a valid XML content. The data are returned to the client in exactly the same form as they were received from the server. The **set** action replaces any previously stored data under this ID.
- **add** - adds new element to the cache under the given ID. This action might be useful for storing data for the opened chat window. The client can add new elements for the chat window, like new messages, icons and so on...
- **remove** - removes the cached element for the given cache ID.

## Cache ID

Cache ID can be an any character string. There might be some IDs reserved for a special cases, like for the Roster content. To avoid any future ID conflicts reserved ID values starts with: **bosh** - string.

There is a default cache ID - an empty string. Thus cache-id attribute can be omitted and then the requests refers to data stored under the default (empty) ID.

## Reserved Cache ID Names

Here is a list of reserved Cache IDs:

- **bosh-roster** - The user roster is cached in the Bosh component in exactly the same form as it was received from the core server. The Bosh Cache might or might not do optimizations on the roster like removing elements from the cached roster if the roster **remove** has been received or may just store all the roster requests and then send them all to the client. There is a one mandatory optimization the Bosh Cache must perform. It must remember the last (and only the last) presence status for each roster item. Upon roster retrieving from the cache the Bosh component must send the roster item first and then the presence for the item. If the presence is missing it means an offline presence. If the roster is small it can be sent to the client in a single packet but for a large roster it is recommended to split contact lists to batches of max 100 elements. The Bosh component may send all roster contacts first and then all presences or it can send a part of the roster, presences for sent items, next part of the roster, presences for next items and so on.
- **bosh-resource-bind** - The user resource bind is also cached to allow the client quickly retrieve information about the full JID for the established Bosh session.

---

# Chapter 9. Old Stuff

This contains sections on old features, or information pertaining to old builds of Tigase. It is kept here for archival purposes.

---

# Chapter 10. Tigase DB Schema Explained

The schema basics, how it looks like and brief explanation to all rows can be found in the schema creation script [<https://projects.tigase.org/projects/tigase-server/repository/revisions/master/entry/database/mysql-schema-4-schema.sql>]. However, this is hardly enough to understand how it works and how all the data is accessed. There are only 3 basic tables which actually keep all the Tigase server users' data: **tig\_users**, **tig\_nodes** and **tig\_pairs**. Therefore it is not clear at first how Tigase's data is organized.

Before you can understand the Tigase XMPP Server database schema, how it works and how to use it, is it essential to know what were the goals of it's development and why it works that way. Let's start with the API as this gives you the best introduction.

Simplified access can be made through methods:

```
void setData(BareJID user, String key, String value);
String getData(BareJID user, String key);
```

And more a complex version:

```
void setData(BareJID user, String subnode, String key, String value);
String getData(BareJID user, String subnode, String key, String def);
```

Even though the API contains more methods, the rest is more or less a variation of presented above. A complete API description for all access methods is available in JavaDoc documentation in the `UserRepository` [<https://projects.tigase.org/projects/tigase-server/repository/entry/trunk/src/main/java/tigase/db/UserRepository.java>] interface. So we are not going into too much detail here except for the main idea.

Tigase operates on `<*key*, value>` pairs for the individual user data. The idea behind this was to make the API very simple and also at the same time very flexible, so adding a new plugin or component would not require a database schema change, adding new tables, or conversion of the DB schema to a new version.

As a result the **UserRepository** interface is exposed to all of Tigase's code, mainly the components and plugins (let's call all of them modules). These modules simply call set/get methods to store or access module specific data.

As plugins or components are developed independently it may easily happen that developer choses the same key name to store some information. To avoid key name conflicts in the database a *node* concept has been introduced. Therefore, most modules when set/get key value they also provide a subnode part, which in most cases is just XMLNS or some other unique string.

The *node* thing is a little bit like directory in a file system, it may contain subnodes which makes the Tigase database behave like a hierarchical structure. And the notation is also similar to file systems, you use just / to separate node levels. In practice you can have the database organized like this:

```
user-name@domain ---> (key, value) pairs
    -|
    roster --->
        -|
        item1 ---> (key1, value1) pairs.
        -|
        item2 ---> (key1, value1) pairs.
```

So to access item's 1 data from the roster you could call method like this:

```
getData("user-name@domain", -"roster/item1", key1, def1);
```

This is huge convenience for the developer, as he can focus on the module logic instead of worrying about data storage implementation and organization. Especially at the prototype phase it speeds development up and allows for a quick experiments with different solutions. In practice, accessing user's roster in such a way would be highly inefficient so the roster is stored a bit differently but you get the idea. Also there is a more complex API used in some places allowing for more direct access to the database and store data in any format optimized for the scenario.

Right now such a hierarchical structure is implemented on top of SQL databases but initially Tigase's database was implemented as an XML structure, so it was natural and simple.

In the SQL database we simulate hierarchical structure with three tables:

1. **tig\_users** - with main users data, user id (JID), optional password, active flag, creation time and some other basic properties of the account. All of them could be actually stored in **tig\_pairs** but for performance reasons they are in one place to quickly access them with a single, simple query.
2. **tig\_nodes** - is a table where the hierarchy is implemented. When Tigase was storing data in XML database the hierarchy was quite complex. However, in a SQL database it resulted in a very slow access to the data and a now more flat structure is used by most components. Please note, every user's entry has something called root node, which is represented by *root* string;
3. **tig\_pairs** - this is the table where all the user's information is stored in form of the <key, value> pairs.

So we now know how the data is organized. Now we are going to learn how to access the data directly in the database using SQL queries.

Let's assume we have a user *admin@test-d* for whom we want to retrieve the roster. We could simply execute query:

```
select pval
  from tig_users, tig_pairs
 where user_id = -'admin@test-d' and
        tig_users.uid = tig_pairs.uid and
        pkey = -'roster';
```

However, if multiple modules store data under the key *roster* for a single user, we would receive multiple results. To access the correct *roster* we also have to know the node hierarchy for this particular key. The main users roster is stored under the *root* node, so the query would look like:

```
select pval
  from tig_users, tig_nodes, tig_pairs
 where user_id = -'admin@test-d' and
        tig_users.uid = tig_nodes.uid and
        node = -'root' and
        tig_users.uid = tig_pairs.uid and
        pkey = -'roster';
```

How exactly the information is stored in the **tig\_pairs** table depends on the particular module. For the roster it looks a bit like XML content:

```
<contact jid="all-xmpp-test@test-d" subs="none" preped="simple" name="all-xmpp-tes
```

---

# Chapter 11. Why the most recent JDK?

There are many reasons but the main is that we are a small team working on source code. So the whole approach is to make life easier for us, make the project easier to maintain, and development more efficient.

Here is the list:

- **Easy to maintain** - No third-party libraries are used for the project which makes this project much easier to maintain. This simplifies issues of compatibility between particular versions of libraries. This also unifies coding with a single library package without having to rely on specific versions that may not be supported.
- **Easy to deploy** - Another reason to not use third-party tools is to make it easier for end-users to install and use the server.
- **Efficient development** - As no third-party libraries are used, Tigase needs either to implement many things on its own or use as much as possible of JDK functionality. We try to use as much as possible of existing library provided with JDK and the rest is custom coded.

What features of JDK-1.5 are critical for Tigase development? Why I can't simply re-implement some code to make it compatible with earlier JDK versions?

- **Non-blocking I/O for SSL/TLS** - This is functionality which can't be simply re-implemented in JDK-1.4. As the whole server uses NIO it doesn't make sense to use blocking I/O for SSL and TLS.
- **SASL** - This could be re-implemented for JDK-1.4 without much effort.
- **Concurrent package** - This could be re-implemented for JDK-1.4 but takes a lot of work. This is a critical part of the server as it uses multi-threading and concurrent processing.
- **Security package** - There number of extensions to the security package which otherwise would not work as easily with earlier versions of JDK.

I think above list is enough to decide to use JDK-1.5. But why JDK-1.6?

- **LinkedHashMap** - in JDK-1.6 is a basement for the Tigase cache implementation.
- **Light HTTP server** - JDK-1.6 offers built-in light HTTP server which is needed to implement HTTP binding (JEP-0124) and HTTP user interface to monitor server activity and work with the server configuration.



---

# Chapter 12. Generating Tigase Installer

To generate installer:

1. Install chosen version of IzPack [<http://izpack.org/>] including source code.
2. In order to compile custom Tigase panels you need to first compile IzPack [<http://izpack.org/>] classes. You can use the included `build.xml` which is in the `src` directory of IzPack [<http://izpack.org/>] install. Just enter this dir and type:

```
ant all
```

1. Depending on the IzPack version classes will be compiled directly into the `src/lib` directory or `_build` directory of IzPack [<http://izpack.org/>]. You may need to tweak the `build.xml` file which is in the same dir as the `readme` and point to the directory where IzPack [<http://izpack.org/>] compiled classes reside.

```
<!-- fragment --->
<classpath>
  <pathelement location="java"/>

  <!-- tweak below fragment --->
  <pathelement location="${installer.path}/_build"/>

  <pathelement location="${installer.path}/bin/panels/TargetPanel.jar"/>
</classpath>
```

1. Make sure that the `bin/panels` directory of IzPack [<http://izpack.org/>] is writable by `generate-installer.sh` script. Compiled custom panels will be placed here before running installer compiler.
2. Modify the script `generate-installer.sh`. Change the `IZPACK_DIR` variable to point to the IzPack [<http://izpack.org/>] installation directory e.g.

```
IZPACK_DIR="/usr/local/IzPack421"
```

3. To start the installation process run the `scripts/generate-installer.sh` file you will find in the main server source code directory. You should start it from the server root dir.
4. Generated files (jar and exe) will be placed in the `packages` dir of Tigase codebase.

---

# Chapter 13. API Description for Virtual Domains Management in the Tigase Server

The purpose of this guide is to introduce vhost management in Tigase server. Please refer to the JavaDoc documentation for all specific details not covered in this guide. All interfaces are well documented and you can use existing implementation as an example code base and reference point. The VHost management files are located in the repository and you can browse them using the project tracker [<https://projects.tigase.org/projects/tigase-server/repository/revisions/master/show/src/main/java/tigase/vhosts>].

Virtual hosts management in Tigase can be adjusted in many ways through the flexible API. The core elements of the virtual domains management is interface `VHostManager` [<https://projects.tigase.org/projects/tigase-server/repository/revisions/master/entry/src/main/java/tigase/vhosts/VHostManager.java>] class. They are responsible for providing the virtual hosts information to the rest of the Tigase server components. In particular to the `MessageRouter` [<https://projects.tigase.org/projects/tigase-server/repository/revisions/master/entry/src/main/java/tigase/server/MessageRouter.java>] class which controls how XMPP packets flow inside the server.

The class you most likely want to re-implement is `VHostJDBCRepository` [<https://projects.tigase.org/projects/tigase-server/repository/revisions/master/entry/src/main/java/tigase/vhosts/VHostJDBCRepository.java>] used as a default virtual hosts storage and implementing the `VHostRepository` [<https://projects.tigase.org/projects/tigase-server/repository/revisions/master/entry/src/main/java/tigase/vhosts/VHostRepository.java>] interface. You might need to have your own implementation in order to store and access virtual hosts in other than Tigase's own data storage. This is especially important if you are going to modify the virtual domains list through systems other than Tigase.

The very basic virtual hosts storage is provided by `VHostItem` [<https://projects.tigase.org/projects/tigase-server/repository/revisions/master/entry/src/main/java/tigase/vhosts/VHostItem.java>] class. This is read only storage and provides the server a bootstrap vhosts data at the first startup time when the database with virtual hosts is empty or is not accessible. Therefore it is advised that all `VHostItem` [<https://projects.tigase.org/projects/tigase-server/repository/revisions/master/entry/src/main/java/tigase/vhosts/VHostItem.java>] implementations extend this class. The example code is provided in the `VHostJDBCRepository` [<https://projects.tigase.org/projects/tigase-server/repository/revisions/master/entry/src/main/java/tigase/vhosts/VHostJDBCRepository.java>] file.

All components which may need virtual hosts information or want to interact with virtual hosts management subsystem should implement the `VHostListener` [<https://projects.tigase.org/projects/tigase-server/repository/revisions/master/entry/src/main/java/tigase/vhosts/VHostListener.java>] interface. In some cases implementing this interface is necessary to receive packets for processing.

Virtual host information is carried out in 2 forms inside the Tigase server:

1. As a **String** value with the domain name
2. As a **VHostItem** [<https://projects.tigase.org/projects/tigase-server/repository/revisions/master/entry/src/main/java/tigase/vhosts/VHostItem.java>] which contains all the domain information including the domain name, maximum number of users for this domain, whether the domain is enabled or disabled and so on. The JavaDoc documentation contains all the details about all available fields and usage.

Here is a complete list of all interfaces and classes with a brief description for each of them:

1. **VHostManagerIfc** [<https://projects.tigase.org/projects/tigase-server/repository/revisions/master/entry/src/main/java/tigase/vhosts/VHostManagerIfc.java>] - is an interface used to access virtual hosts information in all other server components. There is one default implementation of the interface: **VHostManager**.
2. **VHostListener** [<https://projects.tigase.org/projects/tigase-server/repository/revisions/master/entry/src/main/java/tigase/vhosts/VHostListener.java>] - is an interface which allows components to interact with the **VHostManager**. The interaction is in both ways. The **VHostManager** provides virtual hosts information to components and components provide some control data required to correctly route packets to components.
3. **VHostRepository** [<https://projects.tigase.org/projects/tigase-server/repository/revisions/master/entry/src/main/java/tigase/vhosts/VHostRepository.java>] - is an interface used to store and load virtual domains list from the database or any other storage media. There are 2 implementations for this interface: **VHostConfigRepository** [<http://projects.tigase.org/server/trac/browser/trunk/src/main/java/tigase/vhosts/VhostConfigRepository.java>] which loads vhosts information for the configuration file and provides read-only storage and - **VHostJDBCRepository** class which extends **VHostConfigRepository** [<http://projects.tigase.org/server/trac/browser/trunk/src/main/java/tigase/vhosts/VhostConfigRepository.java>] and allows for both - reading and saving virtual domains list. **VHostJDBCRepository** is loaded as a default repository by Tigase server.
4. **VHostItem** [<https://projects.tigase.org/projects/tigase-server/repository/revisions/master/entry/src/main/java/tigase/vhosts/VHostItem.java>] - is a class which allows for keeping all the virtual domain properties. Sometimes the domain name is not sufficient for data processing. The domain may be temporarily disabled, may have a limited number of users and so on. Instances of this class keep all the information about the domain which might be needed by the server components.
5. **VHostManager** [<https://projects.tigase.org/projects/tigase-server/repository/revisions/master/entry/src/main/java/tigase/vhosts/VHostManager.java>] - the default implementation of the **VHostManagerIfc** interface. It provides components with the virtual hosts information and manages the virtual hosts list. Processes ad-hoc commands for reloading, updating and removing domains.
6. **VHostConfigRepository** [<https://projects.tigase.org/projects/tigase-server/repository/revisions/master/entry/src/main/java/tigase/vhosts/VhostConfigRepository.java>] - a very basic implementation of the **VHostRepository** [<http://projects.tigase.org/server/trac/browser/trunk/src/main/java/tigase/vhosts/VHostRepository.java>] for loading domains list from the configuration file.
7. **VHostJDBCRepository** [<https://projects.tigase.org/projects/tigase-server/repository/revisions/master/entry/src/main/java/tigase/vhosts/VHostJDBCRepository.java>] - the default implementation of the **VHostRepository** [<http://projects.tigase.org/server/trac/browser/trunk/src/main/java/tigase/vhosts/VHostRepository.java>] loaded by Tigase server. It allows to read and store virtual domains list in the database accessible through **UserRepository**.

---

# Chapter 14. Stanza Limitations

Although XMPP is robust and can process stanzas of any size in bytes, there are some limitations to keep in mind for Tigase server.

Please keep these in mind when using default Tigase settings and creating custom stanzas.

- Limit to number of attributes of single element = 50 attributes
- Limit to number of elements = 1024 elements
- Limit to length of element name = 1024 characters
- Limit to length of attribute name = 1024 characters
- Limit to length of attribute value = 10240 characters
- Limit to length of content of single element CDATA = 1048576b or 1Mb

These values may be changed.

**Note that these limitations are to elements and attributes that may be within a stanza, but do not limit the overall stanza length.**

## Escape Characters

There are special characters that need to be escaped if they are included in the stanza to avoid conflicts. The rules are similar to normal XML escaping. The following is a list of characters that need to be escaped and what to use to escape them:

&	&amp;
<	&lt;
>	&gt;
"	&quot;
'	&apos;