System Development
for Java
with the z2-Environment

Version 2.6

Last updated on:
September 17, 2018

Copyright by
ZFabrik Software GmbH & Co. KG
Walldorf, Germany

1. Introduction......2
1.1. What is the z2-Environment......3
1.2. Online resources......3
2. Installing and Understanding a Z2 Home......3
2.1. The z2-base Distribution......4
2.2. Home install in 2 minutes......4
2.3. Folder Structure of a Z2 Home......5
2.4. Starting and synchronizing......6
2.5. Important System Properties......7
2.6. The Base Repository......9
2.7. Worker Processes......9
3. Anatomy......10
3.1. Working on-demand......10
3.1.1. The Resource Management System......11
3.1.2. Embedded runtime and the Main Runner......11
3.2. Components and Component Repositories......12
3.2.1. Synchronization with updates......14
3.2.2. File system support......15
3.2.3. Subversion support......15
3.2.4. Git support......15
3.2.5. Maven Repository Support......16
3.2.6. Component Types and Component Factories......18
3.2.7. Dynamic Component Properties......18
3.2.8. Using Dynamic Property Expressions with JEXL3......20
3.2.9. Java Naming and Directory Interface (JNDI) support......21
3.2.10. Link Components and Component Linking......21
3.3. Unit of Work and transaction management......22
4. Constructing a Z2 system......23
4.1. Add-ons......24
5. Developing with the z2-Environment......24
5.1. A note on what JDK to use......24
5.2. Workspace development using the Dev Repository......25
5.2.1. Recommended folder structure......25
5.3. The Eclipsoid Plug-in......26
5.4. Using the Offline Mode......27
5.4.1. How the offline mode is used.......28
5.4.2. Enabling Offline mode at start time......28
5.4.3. Toggling Offline Mode during Development......28
5.5. In-container unit testing using Z2 Unit......28
5.6. Retrieving jars from Z2......28
6. Enhanced Basis Features......29
6.1. The Binary Hub Component Repository......29
6.2. The Gateway for Zero-Downtime-Upgrades......30
7. Component type reference......30
7.1. Core component properties......30
7.2. “Any” components (core)......31
7.3. Component Factories (core)......31
7.4. Data Source components (z2-base)......32
7.4.1. Data Source Specific Configuration......33

7.4.2. Data Source Types....33
7.5. File system component repositories (core)....34
7.6. GIT component repositories (core)....35
7.7. Home Layouts....36
7.8. Java components (core)....37
7.8.1. Classloaders....37
7.8.2. Includes....38
7.9. JUL configurations (z2-base)....41
7.10. Link Components....41
7.11. Log4J configurations (z2-base)....42
7.12. Main Program Components (core)....42
7.13. Maven component repositories (z2-base)....43
7.14. Maven component repository fragments (z2-base)....45
7.15. Subversion component repositories (core)....45
7.16. System States (core)....46
7.17. Web applications (z2-base)....48
7.18. Web servers (z2-base)....49
7.19. Worker Processes (z2-base)....50
7.19.1. System Property Propagation from Home to Worker Process....51
7.19.2. Special considerations when specifying VM options using blanks and quotes....52

1. Introduction

Developing applications for the Java platform typically involves a number of steps that are needed to be performed before execution. At the least it is required to run a Java compiler that turns Java source code into executable byte code.

Coding environments like the Eclipse or IntelliJ Integrated Development Environments do a great job of hiding these operations to the user for simple applications. When applications become more complex though, modularization needs and team development approaches ask for more sophisticated tools to make sure that growing system complexity can be managed.

The traditional development approach for Java applications resembles that of Desktop applications. Apart from compiling source code into an executable representation, execution of additional operations, the *build process*, needs to be completed before execution. This includes resolution of module dependencies and packaging of generated as well as retrieved artifacts into some *deployable* file format that will be installed (*deployed*) in some execution environment. For Java applications the latter is either a standalone Java Virtual Machine (JVM) or an application server.

For Desktop applications that can assume very little about their execution environment it is a necessity to be completely self-contained and easily distributable in form of some bundle of files.

The situation for Intranet and Internet applications is different. A major part of the lifecycle of a business application installation consists of change: ongoing development, customization, extension, and repair.

All but the most trivial business applications are a composition of subsystems that make strong assumptions on the presence and behavior of other subsystems – all together forming a software system that offers a wide range of access methods, performs recurring background jobs, integrates with other systems in whole landscape of systems, and operates over a shared and evolving data asset.

Platforms that are heavily geared towards development and operation of business applications, such as SAP's ABAP environment or Oracle's PL/SQL platform therefore take a different view. Instead

of mimicking the concept of a most generic operating system that runs largely independent, locally *installed* binary applications, the focus of these environments is to perform the functions of a large highly interconnected software system at scale, agnostic to the single machine, with as little local configuration as possible. Instead of being the end of a tool chain that is a mere executor of binary code in some undecipherable interplay, a centrally defined, customizable and extensible system definition in the form of source code and configuration that is executable without build process complexities by arbitrarily many machine nodes running the platform is crucial to managing software life cycle complexity at scale.

The z2-Environment brings these qualities to the world of Java applications. We call it the *system-centric* approach.

1.1. What is the z2-Environment

Practically speaking the z2-Environment is a Java-based runtime environment that knows how to update itself from source code and configurations stored in repositories of various technologies, including source control systems like Git and Subversion or just a plain old file system.

Z2 defines an extensible component and modularization model that, based on few basic paradigms and interfaces, allows to construct full-blown modular application systems.

The z2-Environment can be used to build Java EE Web applications as well as standalone Java applications without restricting the use of third-party libraries and popular frameworks like the Spring Framework, Hibernate/JPA, and many more.

Z2 is strictly implemented on Java:

- Versions before 2.4 require Java 6 and support Java 6 and Java 7 language levels.
- As of version 2.4, Z2 requires Java 8.
- As of version 2.6, Z2 requires Java 9.

When using a higher, future Java version, version 2.5 will fall back to assuming Java 8 language level while version 2.6 will assume Java 9 or Java 10 language level.

1.2. Online resources

Z2 can be installed and tried out in a matter of minutes. There are various how-tos and samples available that explain and demonstrate the use of Z2.

Please visit the Wiki at

http://redmine.z2-environment.net/projects/z2-environment/wiki

to start your exploration on all practical matters.

This documentation and high-level information as well as latest news can be found on the official Z2 home page at

http://www.z2-environment.eu/home.

2. Installing and Understanding a Z2 Home

In order to access component repositories and to implement its component and modularization model at runtime, the elementary capabilities of Z2 need to be installed as a normal Java program in binary form. The most fundamental features of Z2 are implemented in the *z2 core*. It's source code and its simple build script, including instructions, can be found on the Wiki site. We call a local

installation of a z2-core a *z2 Home*.

In its basic form, the z2 core knows little more than running Java main programs from source in a modular context. In order to turn a z2 home into a node of a capable system we need to connect it to repositories defining further component types, libraries, and applications.

This is done by declaring component repository components as will be explained below. Choosing remote repositories will give you a system that is centrally defined and scales easily with consistent updates.

Choosing only local repositories will leads to system that can be maintained and distributed like a Scripting language application, albeit being implemented using Java.

2.1. The z2-base Distribution

Starting with version 2.6, when you download z2 for installation, you are encouraged to download the **z2-base distribution** from the Web site. The z2-base distribution does not only contain the z2 core but also a locally configured **z2-base.base** repository in the **base** folder (see below), providing the Jetty Web Container and other components that are useful for most anything you might want to do, without requiring remote access to the z2-base Git repositories.

The alternative is to use the **z2-core distribution** that only contains the z2-core binaries and has a remote-configured z2-base repository. For larger applications where the need for z2 home updates should be minimized this would be the right choice.

To sum it up:

Distribution	Contains	When to use
z2-base	The z2 core binaries and a locally configured z2-base.base repository.	Getting started. Needs no remote access to anywhere and can be easily extended using local modules (see #develop).
z2-core	The z2 core binaries and a remote configured z2-base.base repository.	Updating or setting up a distributed system that is configured and extended using remote repositories.

2.2. Home install in 2 minutes

Generally you install a z2 Home by downloading and unpacking a z2 base distribution from the Web site. Distribution ZIP an TAR archives are named including the distribution base name (e.g. z2-base), the version branch, as well as the actual build time stamp.

For example, from a linux command line prompt you might run:

```
mkdir install
cd install
wget http://download.z2-environment.net/z2/z2-base-v2.6.tar.gz
tar xf z2-base-v2.6.tar.gz
z2-base.core/bin/gui.sh
```

installing a z2 Home at z2-base.core in the install folder and starting the development graphical user

interface.

2.3. Folder Structure of a Z2 Home

Typically a home installation folder has the following file and folder structure:

bin
This folder holds the actual z2 start code and shell scripts. This is from where you run z2.

local
This folder holds all core modules that have been pre-compiled as part of the z2-core build. Normally you do not need to touch anything in here.

base
When using the z2-base distribution this folder holds the complete z2-base.base modules. The z2-base.base repository is the application foundation on z2 and holds the Jetty web container and some fundamental libraries and applications.

config
The config folder is another z2 component repository meant to be used to define modules that tweak or add configuration to z2, in particular other sources of modules. By default there is a repos module that binds the base repo (see one row up).

licenses
This folder holds a reference or copy of all licenses per library and module that is part of the distribution you installed. A browsable version of this content can be found on release page on the Wiki. For example, for version 2.6, go here:

https://redmine.z2-environment.net/projects/z2-environment/wiki/Third_Party_Licenses

Note: To our knowledge, there are no non-business-friendly OSS licenses, but you need to check for yourself.

LICENSE.txt and THIRD-PARTY-LICENSES.txt

The license you retrieved z2 under, the Apache 2 Open Source Software license, as well as information on how to find out about third-party licenses, just as above.

work

During runtime z2 needs to store essentially transient data like compilation results or other cached data. While a running z2 may not be able to handle a missing work folder, it is generally ok and sometimes even helpful to remove all transient work data when z2 is down. This folder is generated during runtime and not part of the distribution.

data

The data folder is used by applications that need to maintain local, file-system stored data, for longer. In short: The data folder should not be deleted casually and, depending on your application, it may be advisable to have it point to particularly reliable storage.

logs

The place to write log files to. Z2 itself writes its log output into logs/home_0_0.log.

A few files are important in the bin folder:

runtime.properties

The properties stored in runtime.properties are loaded by the home process and all worker processes into the respective JVM system properties. See also #systemProps.

launch.properties

A z2 home can be started in different "modes". This is a convenience feature to simplify the application of various Virtual Machine settings for the home process (many of which propagate to worker processes – including debug settings).

A typical launch.properties file looks like this:

```
#
# alternative VM profiles for the home process
# VM opts
# default
#
home.vmopts=\
   -Xmx32M -cp z.jar \
   -Dcom.sun.management.config.file=management.properties \
   -Djava.util.logging.config.file=logging.properties \
   -Dcom.zfabrik.home.concurrency=5 \
   -Dcom.zfabrik.home.layout=environment/home \
   -Dcom.zfabrik.dev.local.workspace=../../../../.. \
   -Dcom.zfabrik.dev.local.repo=../../work/repos/dev \
   -Dcom.zfabrik.mode=development

# override when -mode debug
home.vmopts.debug=-Xdebug -Xnoagent -
Xrunjdwp:transport=dt_socket,suspend=n,server=y,address=5000 -
Dworker.debug=true

# override when -mode verbose
home.vmopts.verbose=-verbose:gc -XX:+PrintClassHistogram
```

2.4. Starting and synchronizing

In order to start Z2 in server mode, change into the folder **Z2_HOME/bin**

If you prefer a simple console view, you can start the z2 Environment by issueing the command **./go.sh** (on Linux/Mac OS) or **go.bat** (on Windows).

There are several options you may use to alter the default behavior. Most notably, if you start using

```
./go.sh -mode:debug
```

the z2 Environment will start with debug settings (see launch.properties above).

If you start using

```
./go.sh - -np
```

the home process will end up showing an input prompt – which is favorable to running as background process (e.g. using nohup or via an init script on Linux). And of course, if you run

```
./go.sh -mode:debug - -np
```

you will get both.

The general syntax is

```
./go.sh <parameters for the launcher> - <parameters for the home process>
```

where the launcher is the small program that computes the actual Java command line as indicated in the previous section.

When running the z2 Environment locally, in particular during application development, it is convenient to have a graphical user interface (GUI). Adding the **gui** option achieves just that. For example

```
./go.sh -mode:debug - -gui
```

starts the home process with a Java GUI that allows to scroll through the home and worker processes console output and to manage synchronizations as well as the current list of worker processes.

The gui shell command is a shortcut that spares you the **gui** option. I.e

```
./gui.sh
```

is a short version of **./go.sh - -gui.**

2.5. Important System Properties

Some system properties can be set on the command line (or launch.properties) or for all z2 processes (e. g. in runtime.properties, see #runtimeProperties).

Property name	Meaning
com.zfabrik.mode	Set to "development" for development mode. Some features, such as test code and switchboard, will be

	ignored unless in development mode. This applies to the home process and worker processes.
com.zfabrik.offline	If set to true, enables the offline mode (see #offlineMode). Can be toggled on the GUI. Will be automatically propagated to worker processes at synchronization.
com.zfabrik.config	Name of the system config file to be loaded by all processes. Defaults to runtime.properties. This applies to the home process and worker processes.
com.zfabrik.home	The z2 Home folder, i.e. where the z2 core is installed. If not specified when starting Z2, this property is determined by checking for the environment variable **Z2_HOME**. If that is not set either, it defaults to "..", one up from the current work folder, which is the installation folder, if you are in the **bin** folder of standard z2 Home folder structure.
com.zfabrik.home.layout	Home layout to start by the home process. This determines the worker processes to start. This applies only to the home process.
com.zfabrik.java.level	Java language level used for compilation. Can be 6,7,8. Determined by the Java runtime in use.
com.zfabrik.home.autoVerifyInterval	Interval in seconds after which the home process will run a verification (an attempt to attain all target states) again. Defaults to none (undefined). This applies only to the home process.
com.zfabrik.dev.local.workspace	Setting of the development repository defining where to check for "armed" modules. See also #devRepo. Relative to current working folder, which is typically the **Z2_HOME/bin**. Typically set to "../../..", i. e. in the direct neighborhood of **Z2_HOME**. This applies only to the home process.
com.zfabrik.dev.local.depth	Setting of the development repository defining how *deep* the Dev Repo scans for LOCAL files. Depth is measured in path distance from the roots set via the system property **com.zfabrik.dev.local.workspace**. Defaults to 3. See also #devRepo.
com.zfabrik.repo.mode	General repository access mode. If set to "**relaxed**", repositories will attempt to work on previously cached resources in case of technical repository access failures during synchronization. If set to "**strict**" (which is the default), technical failures will lead to a failure in synchronization, even if local resources are available. This feature is useful for offline or bad connectivity situations.

Note see also #workerProcesses on how to make sure properties are propagated to worker processes. More important system properties, including some that will be set by z2 at runtime, can be found in the Foundation class.

2.6. The Base Repository

The Z2 core that gets cloned (or checked out) to create a Z2 Home contains exactly what is needed to be able to bootstrap a running environment. All further definitions, code, and configuration is retrieved via additional component repositories that are typically accessed remotely.

The starting point, from the perspective of the Z2 core is the so-called *Base Repository*.

The Base Repository is defined in **Z2_HOME/local/com.zfabrik.boot.config/baseRepository.properties.** By default the Base Repository points to the **z2-base.base** repository hosted on z2-environment.net.
When you create your own system, this is an important customization point. We will get to that in Constructing a Z2 system after we have learned a about Components and Component Repositories.

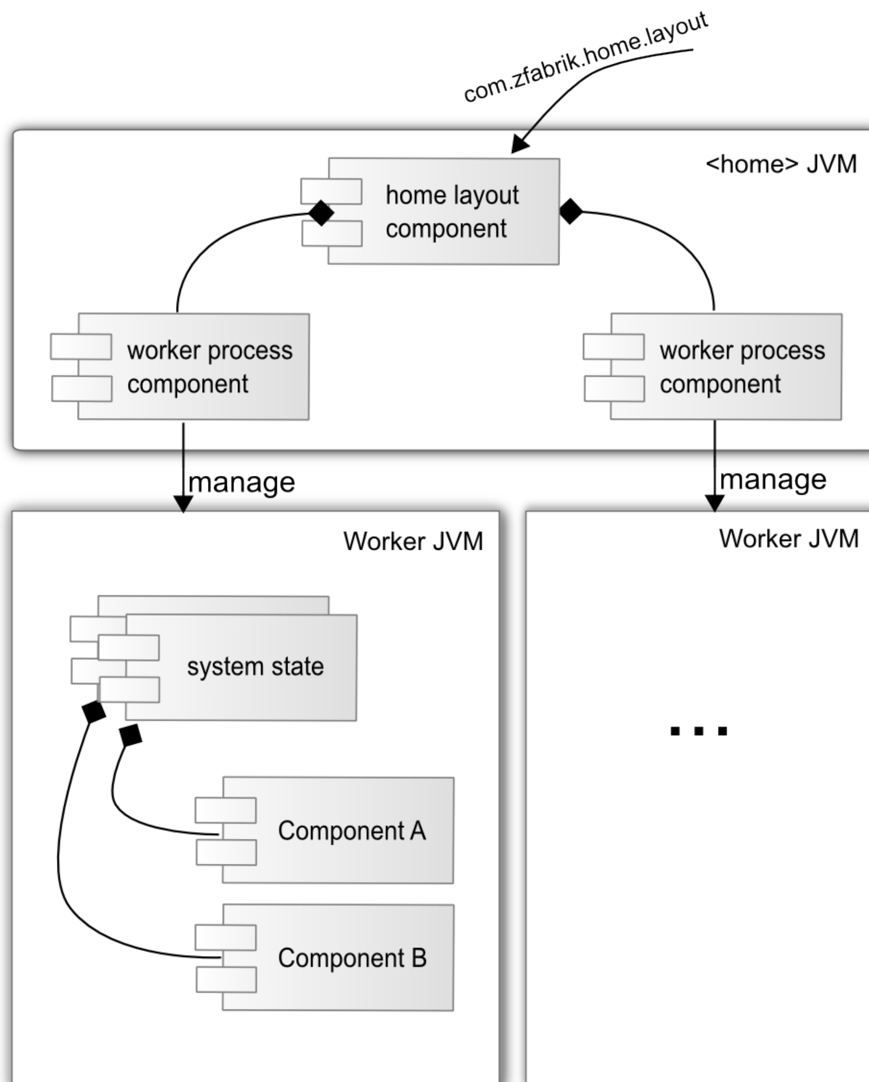
2.7. Worker Processes

As indicated above the z2-Environment can manage further JVM processes to better support heterogeneous load scenarios without compromising the ability to apply updates consistently nor the stability of the home process

Worker processes are, from a home process perspective, regular z2 components. See also the documentation of the component type **com.zfabrik.worker** below. What makes worker processes differ is their virtual machine configurations and the set of target states to attain (see **com.zfabrik.systemState** below). Target states on the other hand group components, e.g. web applications to be started and kept alive.

Worker processes are typically loaded when maintaining a home layout (see below for **com.zfabrik.homeLayout**). A home layout is simply a list of worker process component names that identify the worker processes to be started when (re-) loading the home layout

The usefulness of home layouts becomes clear when understanding that the home process always loads the home layout specified by the system property **com.zfabrik.home.layout**. This way, completely different worker process combinations combinations from the very same shared configuration store can be achieved by starting home processes with different values of the system property **com.zfabrik.home.layout.**



3. Anatomy

3.1. Working on-demand

Much of the goodness of the Z2-Environment comes from the fact that it has a pervasive on-demand architecture. That means that whenever the runtime binds resources it is for a clear and understandable reason, either because a need (a dependency) has been declared or the specific task at hand requires so.

While that sounds trivial, it is not so. Unlike implicit "start of everything deployed" or "everything in a list" approach that many application servers implement, binding of runtime resources from the potentially large pool of components available in a component repository (see below) happens strictly as required based on target states configuration – which eventually translates to simplified on-demand operations of large scale out scenarios with heterogeneous node assignments.

The sibling of the load-on-demand approach is the unload-on-invalidity approach. When repository definitions have been updated, in development but also in production scenarios, Z2 runtimes can adapt to the changes made. That requires to understand what component definitions have become invalid and to "unload" these from memory. Because of the modular nature of components and the heavy re-use of resource, invalidation of one component typically implies that others, dependant

components have implicitly become invalid as well.

For example, a change in an API defined in some Java component may imply that web applications have to be restarted.

The abstraction for resources that have dependent resources is the Resource Management system of Z2.

3.1.1. The Resource Management System

The Resource Management system is at the heart of the Z2 runtime. Essentially anything that binds runtime memory or represents components is internally modeled as extensions of the Resource class (see Resource).

Resources represent any kind of abstraction that may be made available for some time and that may have dependencies onto other abstract resources, such as cache regions, applications, etc. In particular z2 components are resources.

Resources are provided by Resource Providers that establish a namespace of resources. One of which is the components resource provider that uses the component factory mechanism to delegate resource construction further.

A resource can be asked for objects implementing or extending any given Java type using the IResourceHandle interface. For components, the IComponentsLookup.lookup method is simply a delegating facade to that.

A complete description of the resource ranagement system is beyond the scope of this section. Please see the documentation of the com.zfabrik.resources packages in the core API Javadocs.

3.1.2. Embedded runtime and the Main Runner

The Z2 environment can be used as a multi-process server environment, which is what we looked at above, or embedded.

Running it embedded simply means to initialize the resource management system and component system from within another JVM process.

This execution mode can be handy for various purposes:

- You can use it to run "Main" programs (see XYZ) that are defined in some component repository from the command line w/o worrying about local build environments (and dependency resolution)
- Sometimes you have no control over the execution mode because your code has been started by some other infrastructure. This is for example true for Hadoop Map-Reduce jobs. In that case the Hadoop Map-Reduce implementation starts tasks from a simple JAR file on some machine. Using the embedded mode we can execute Map-Reduce jobs defined in component repositories, without complicated job assembly into a hadoop job jar.

To facilitate the embedded mode, the Z2 home provides the **z_embedded.jar** in **Z2_HOME/bin**.

Pre-requisite to using Z2 in an embedded way is to have a Z2 home installation in file system reach. That home installation will be used to cache component repository content and binaries – i.e. it is essential to actually implement Z2.

When you open a console and run

```
java -jar z_embedded.jar
```

you will get

```
z2 MainRunner: Main method execution of z2 components.

Usage:   java -DcomponentName=<component name> -jar z_embedded.jar <arg1>
<arg2>...

Note: Make sure to either set a Z2_HOME environment variable pointing to
the relevant z2 home installation or specify the system property
com.zfabrik.home when calling the MainRunner:

java -DcomponentName=<component name> -Dcom.zfabrik.home=<home folder> -
jar z_embedded.jar <arg1> <arg2>...
```

explaining the most direct way of using the embedded mode.

There are several Main programs that come with the z2-base system. For example a tool to retrieve binaries from Z2: **com.zfabrik.dev.util/jarRetriever**

Running:

```
java -DcomponentName=com.zfabrik.dev.util/jarRetriever -jar z_embedded.jar
-out test com.zfabrik.dev.util/java
```

Retrieves the binaries of the Java component **com.zfabrik.dev.util/java** into the folder **test.** See also com.zfabrik.dev.util (note that the environment variable Z2_HOME was expected. Otherwise use the system property **com.zfabrik.home** to specify the home path).

for the command line and more details. The other way of embedded execution is via the ProcessRunner class (in the core API).

3.2. Components and Component Repositories

Everything you ever touch that the z2 Environment is supposed to understand is organized in *Components.* Z2 is built around the concept of named components that are defined in a well-defined repository structure. The level of understanding of resources that are used to implement some functionality in z2 is essential so that z2 understands when resources have been modified and corresponding runtime objects have become invalid and so that z2 is extensible by new semantics, that is new types of components.

More specifically the term Component translates in z2 to runtime objects that implement semantics according to a *Component Type*, have a well-defined, location-derived name, and are declared by a set of properties and optionally any kind of file type resources – e.g. holding the files of a Web Application.

Component properties are accessible within z2 via the IComponentsManager interface providing IComponentDescriptor instances.

Component properties are typically declared via *.properties file resources in stores backing component repositories such as Git, Subversion, or a file system.

Component properties can be declared in a static, non-computed way but also with some dynamic handling as descibed in Dynamic Component Properties.

Even more specifically, most existing Component Repositories implement the following folder structures that define components as shown in the right column:

… <module>/<local>.properties ...	Defines component **<module>/<local>** of type of value of the property **com.zfabrik.component.type** as set in the property file **<local>.properties.**
... <module>/<sub>/ z.properties <file/folder> <file/folder> … ...	Defines component **<module>/<sub>** of type of value of the property **com.zfabrik.component.type** as set in the property file **z.properties.** The component has furthermore all resources defined in all files and folders under **<sub>.** These can be accessed using **IComponentsManager.INSTANCE.retrieve(<folder>/<sub>)**

The module taxonomy has no little technical meaning to Z2. There is no internal Object representing a module. Via conventions however the module (as being the path without its last segment) has a rather prominent function:

- Typically the module matches the development project granularity
- The resolution of Java resources for a component defaults to <module>/java (see Java components).
- If two component repositories define components of an equally named module, no component of that module of the repository with the lower priority will be visible.

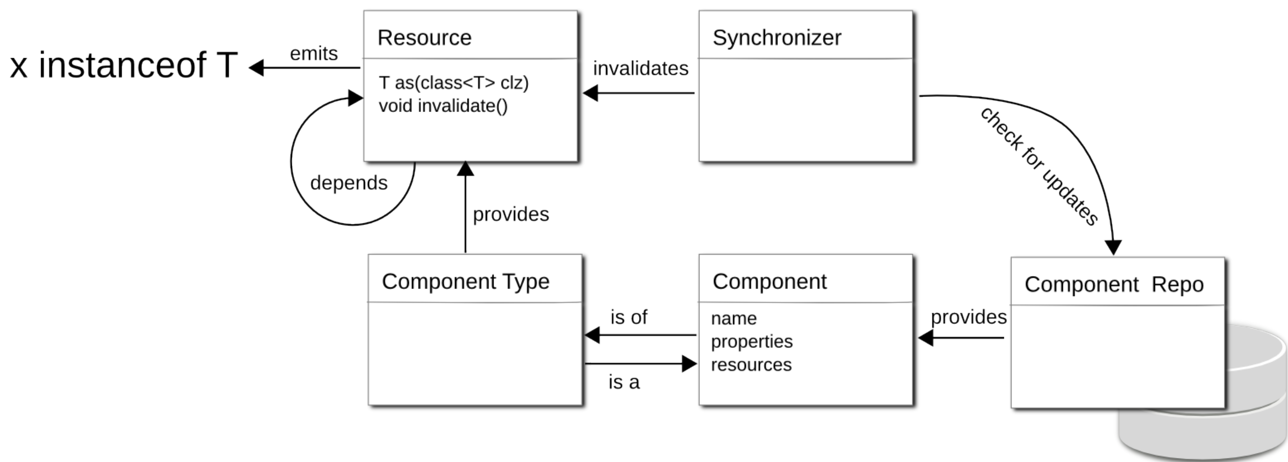
Component repositories define the reality for the z2 environment. So it is important to understand this concept to understand z2.

Component repositories are, of course, declared as components itself. Consequently, component repositories may hold further definitions of component repositories – potentially leading to some reality distortion (aka Bootstrapping) issues – in the rare case you do advanced repository wiring.

When the z2 Environment starts up it has a hard coded knowledge of the *Local Repository* that is stored in **Z2_HOME/local** (see also above). It is the root of the Z2 component universe from a Z2 Home perspective.

The Local Repository also defines the Base Repository, that we briefly touched on in The Base Repository and will meet again in Constructing a Z2 system.

The following diagram is an overview over the few entities that really are the heart of Z2 core – from component repository to resource via synchronization:



The notable exception to the repository structure above is Maven component repositories and – to some extent – the Hub Component Repository. The former derives Java components from Maven repository artifacts. That is, while the underlying structure is completely different, the repository implementation presents it in a Z2 compliant way (see Maven component repositories).

The Hub Component Repository turns a Z2 system into a repository for another Z2 system with the purpose of reducing bandwidth requirements for the original repositories or to not send source code over the wire (see the Hub Component Repository.

3.2.1. Synchronization with updates

At times, frequently when you are developing and less frequently in production, you want your runtimes to get up to date with respect to repository contents. That process is called *Synchronization.* The ability to synchronize with repositories is a particular capability of the z2-Environment and responsible for much of its goodness.

The synchronization process happens in three phases: At first, in the pre-invalidation phase, all component repositories (actually all "synchronizers", but component repositories are generally connected to synchronizers. See also ISynchronizer) are asked to check whether there are updates available and what components (by name) will be affected. In the simplest case, the file system stored component repository, the check will examine folders to find out whether files have changed since the last time it was asked to check.

When that phase has completed, all components that have been identified to be subject of updates will be invalidated. Invalidation is a concept of the Resource Management systerm underlying z2. Loosely speaking it means that a component is asked to let go of all state but its name. Anything that is dependent on repository content or other components it depended on is to be dropped.

In the completion phase of the synchronization, synchronizers are asked to make sure that at the end of the completion phase the runtime has attained operational modes again. That is maybe the most interesting phase, as actions to that end may greatly vary.

For example, the home synchronizer (**com.zfabrik.boot.main/homeSynchronizer**) will simply try to attain the home_up state again.

The worker synchronizer (**com.zfabrik.workers/workerSynchronizer**) will send all invalidations to the worker processes and then ask them to attain their target states again.

Note that synchronizers have a priority and are called in a defined order. So that the worker synchronizer is called before the home synchronizer. As worker processes may have been

invalidated in the second phase, it would be unreasonable to first bring them up again (home synchronizer) just to tell them about invalidations once more.

3.2.2. File system support

The simplest of all built-in component repositories is the file system component repository. All that is required is a file system folder holding component declarations and component resources in a structure as described above. As laid out below, always make sure the repository is started early on by declaring a participation in the system state **com.zfabrik.boot.main/sysrepo_up**.

See File system component repositories for more details on the configuration of file system component repositories.

3.2.3. Subversion support

The popular source control management system Subversion (see www.tigris.org) was the first repository supported by z2 and still shines in many aspects.

In order to add a subversion component repository, declare a component of type **com.zfabrik.svncr** as described in Subversion component repositories.

To avoid problematic licenses, the z2 Environment does unfurtunately not come with the complete built-in Subversion connectivity. Additional configuration steps are required once to complete subservion enabling on your side, as described in the Subversion How-To.

As noted above, it is important to make sure your repository participates in the system state **com.zfabrik.boot.main/sysrepo_up**, i.e. you should add the line

```
com.zfabrik.systemStates.participation=com.zfabrik.boot.main/sysrepo_up
```

to the repository declaration. The URL of the repository should point to a repository folder structure as outlined in Components and Component Repositories. For example, the Base Repository of the z2@base distribution has the URL:

```
http://z2-environment.net/svn/z2-environment/trunk/z2-base.base
```

3.2.4. Git support

The GIT version control system (VCS) is an implementation of a distributed version control system (DVCS). As opposed to centralized VCS, such as Subversion below, in a DVCS users hold a copy (called a *clone*) of the repository content on their local environment, typically the local disk, and can execute all typical modification operations, such as adding files, committing changes, to the Local Repository before sending updates back to a remote repository or retrieving updates from a remote repository.

Currently all framework development for Z2 happens in Git. All results are however available from Subversion repositories as well.

From a Z2 perspective a DVCS has the advantage of giving a slightly easier way of getting your own local repository that is fully under your control. Also moving changes between systems has a built-in solution this way. On the downside, you pay by distributing complete copies of your

system's repository which may turn into a problem once repositories get significantly bigger than what is actually needed for the given scenario. That's why there is an implied tendency for more and smaller repositories when using Git und fewer but larger repositories when using Subversion.

In order to add a Git component repository, declare a component of type **com.zfabrik.gitcr** as described in GIT component repositories .

As noted before, it is important to make sure your repository participates in the system state **com.zfabrik.boot.main/sysrepo_up**, i.e. you should add the line

```
com.zfabrik.systemStates.participation=com.zfabrik.boot.main/sysrepo_up
```

to the repository declaration.

3.2.5. Maven Repository Support

Maven repositories such as Maven Central are a huge source of open-source libraries made available directly by the copyright holder. Maven repositories can be integrated into a Z2 system as component repositories. In fact, most of the samples accessible from the Z2 Wiki as well as prominent add-ons such as the Hibernate and Spring add-on make use of this approach.

The main idea is that based on some root artifacts and some maven remote repository configuration, jar artifacts and dependencies will be made available as Java component in Z2 that can be referenced or included as suits best.

Artefacts in Maven repos have a fully qualified name of the form

```
<groupId>:<artifactId>:<version>
```

or

```
<groupId>:<artifactId>:<packaging>:<version>
```

By default, a jar artifact **<groupId>:<artifactId>:<version>** will result into a Java component of name

```
<groupId>:<artifactId>/java
```

As usual with Maven, if resolution root artifacts and dependencies lead to artifacts of the same packaging, group id, and artifact id but with different versions, a conflict resolution takes place (the higher version number will be used).

By default, all non-optional compile scope dependencies will be resolved. The resulting Java component will have the target artifact as API library and all non-optional compile scope dependencies as public references in their mapped form.

The z2 core will use lazy component class loaders to make sure that use of include libraries has virtually no runtime penalty.

An example configuration of a component repository from a Maven artifact repository may look like this:

```
com.zfabrik.systemStates.participation=com.zfabrik.boot.main/sysrepo_up
com.zfabrik.component.type=com.zfabrik.mvncr

mvncr.repository=mavenDefault
mvncr.priority=200
mvncr.roots=\
    org.springframework:spring-context:4.0.2.RELEASE,\
    org.springframework:spring-aspects:4.0.2.RELEASE,\
    org.springframework:spring-tx:4.0.2.RELEASE,\
    org.springframework:spring-orm:4.0.2.RELEASE,\
    org.springframework:spring-web:4.0.2.RELEASE,\
    org.springframework.security:spring-security-core:3.2.2.RELEASE,\
    org.springframework.security:spring-security-web:3.2.2.RELEASE,\
    org.springframework.security:spring-security-config:3.2.2.RELEASE,\
    org.springframework.security:spring-security-aspects:3.2.2.RELEASE,\
    org.hibernate:hibernate-entitymanager:4.3.4.Final,\
    aopalliance:aopalliance:1.0,\
    org.aspectj:aspectjweaver:1.7.4,\
    org.aspectj:aspectjtools:1.6.9,\

mvncr.excluded=\
    org.jboss.spec.javax.transaction:jboss-transaction-api_1.2_spec

mvncr.managed=\
    commons-logging:commons-logging:1.1.2
```

This configuration would imply that the listed roots and all non-optional compile time references would be added as Java components with mapped references as described above. As an exception however, the artifact **org.jboss.spec.javax.transaction:jboss-transaction-api_1.2_spec** would be excluded. Furthermore, the artifact **commons-logging:commons-logging** would exclusively be used in version 1.1.2. Note: Those modifications of the dependency graph resolution correspond to similar Maven configurations (notably as in <exclusions> and <dependencyManagement>).

See Maven component repositories in the component reference for more details on configuration properties of Maven component repositories.

At times, it is useful to not have all required dependency roots in one component declaration but rather allow some modularization-friendly spread out declaration of dependency roots within a system.

This is achieved by using Fragments of a Maven component repository.

A fragments adds to the dependency graph but does not define where dependencies are retrieved from, so that the requirement for artifacts can be expressed without wiring the actual system to a specific environment. This is how the standard add-ons express their requirements.

Note that having two sets of roots combined and resolved is not equivalent to having to independent Maven Component Repository declaration as version conflict resolution (to the higher version) will always happen within the scope of one Maven component repository. In fact, in most cases having one Maven Component Repository will be the only manageable approach.

In order to add a fragment to a Maven Component Repository declare a component of type **com.zfabrik.mvncr.fragment**.

See Maven component repository fragments in the component reference for more details on configuration properties for fragments.

When running in Development mode, the repository will also provide the source (classifier) artifact if available, so that the Eclipsoid plugins will provide source code attachments to the development environment whenever possible during classpath resolution.

Z2's implementation is based on Eclipse-Aether.

As this subject is not completely trivial it is strongly recommended to check out the samples and explanations on the Wiki. Start with the Maven Component Repository How-to.

As noted before, it is important to make sure your repository participates in the system state **com.zfabrik.boot.main/sysrepo_up**, i.e. you should add the line

```
com.zfabrik.systemStates.participation=com.zfabrik.boot.main/sysrepo_up
```

to the repository declaration.

3.2.6. *Component Types and Component Factories*

Every component in Z2 has a type, declared via the component property com.zfabrik.component.type. As indicated above the component type identifies the semantics of a component, i.e. how to treat it and what you can do with it. For example a web application is of type **com.zfabrik.ee.webapp.** Being of that type implies an expected folder structure for the resources that belong to the web application. Also it implies the ability to be made available via a web server. The semantics of a Java component (of type **com.zfabrik.java**) is obviously completely different.

Component Factories are in charge of implementing the semantics of a component type. In short a whenever a component is requested via the resource management system, the component factory responsible for the respective component type is asked to create an implementation, more specifically a Resource (see The Resource Management System) that implements the actual component.

So, for example, the component factory for web applications knows how to interpret the folder structure of a web application component as the layout of a Java EE web application and how to register this web application with the Jetty web container. The component factory for Java components knows how to check whether code needs to be compiled and how and how to set up class loaders.

3.2.7. *Dynamic Component Properties*

Component properties can be specified to be dynamically evaluated at runtime. I.e. instead of assigning a static string value, input is specified that is processed by custom code before assigned as value to the property.

The most obvious use-case for this feature is to support property values that are derived from system properties, process environment variables, or properties of some other component. Built-in support for these use-cases is provided by the JEXL3 processor based on the Apache JEXL library (http://commons.apache.org/proper/commons-jexl/) version 3. See also #jexl3support for details.

In order to have property values evaluated in a non-static way, the property is extended by a processor type identifier:

```
<property name>:<processor type>=<expression>
```

with a default processor type “plain”.

For example, a propery file,

```
svncr.uri\:plain=http://myserver/svn/myrepo
```

is equivalent to

```
svncr.uri=http://myserver/svn/myrepo
```

(note that the colon “:” character needs to be escaped with a leading backslash “\” in property file syntax so that it is not taken for a separator of name and value).

However,

```
svncr.uri\:JEXL3=env.SVN_REPO+”/myrepo”
```

will evaluate the environment variable **SVN_REPO** and append a string “/myrepo” (see also #jexl3support).

Due to a number of reasons, the use of dynamic property evaluation is subject to various constraints:

- As a principle, when using the IComponentDescriptor API, getProperties() returns processed properties while the method getRawProperties() returns the unprocessed component properties in its original verbatim form.
- Processing of properties happens after retrieving properties from the original or intermediate storage, and in particular evaluation results will not be kept between component loading. That is, at latest after a restart and also after an invalidation due to a synchronization, component expressions will be evaluated again. However, after loading and as long as held onto during runtime nor repeated evaluation will be performed.
- The IComponentsManager findComponents(...) API is using raw component properties to find components based on search conditions. That is, a dynamic expression cannot be used to alter search results for extension points or component type implementations.
- When making use of dynamic properties early in the z2 loading life cycle, in particular in conjunction with component repository definitions and custom component descriptor processor implementations, particular care needs to be paid to the availability of the processor at time of evaluation of component repository declaration.

3.2.8. Using Dynamic Property Expressions with JEXL3

A component descriptor processor with processing type “JEXL3” comes built-in with z2 starting version 2.5. This processor is based on the Apache Commons JEXL library version 3.0 (see

https://commons.apache.org/proper/commons-jexl). All values of properties tagged as of JEXL3 processing style in the original component descriptor source will be considered JEXL expressions.

If evaluating to a non-null value, the expression evaluation result will be set as resulting property value of the same property name. If the expression evaluates to **null**, the corresponding property will not be set in the overall evaluation result.

The following implicit variables are made available by this processor:

Variable name	Definition	Example
system	System.getProperties()	`user dir is ${system["user.dir"]}`
env	System.getenv()	"JAVA_HOME is ${env.JAVA_HOME}"
this	The actual property set in its evaluated form. This may be used to resolve properties that are again computed by some processor. Note that a max evaluation depth of 50 is enforced when resolving properties defined in the property set	this[“com.zfabrik.component.type”]
components	IComponentsManager.INSTANCE	components.getComponent("mymodule/mycomponent").getProperty("myProp")

Using the **this** variable, re-use of definitions within a single descriptor is possible. For example, assuming even the involvement of some other processor *ext*, a property set

```
hostName\:JEXl3=env.hostName
database\:ext=_THE_DB_
ds.prop.url\:JEXl3=`jdbc:derby://${this.hostName}/${this.database}`
```

would be evaluated resolving across processors and with correct substitution in the JEXL expression.

Note that a JEXL expression does not necessarily evaluate to a string object. Neither is the input property value to be processed considered a JEXL string. For example, the expression

```
`Hallo this is ${system["user.name"]}`
```

does indeed evaluate to a string. As does

```
"Hallo this is ${system[\"user.name\"]}"
```

(without processing the ${} expression) and more advanced

```
system["os.name"].startsWith("Linux")? "We are running on a Linux OS" :
"We are running on something else"
```

but not

```
500+10
```

which is indeed an integer.

See also https://commons.apache.org/proper/commons-jexl/reference/syntax.html for more syntax information.

3.2.9. Java Naming and Directory Interface (JNDI) support

Components in the z2-Environment may be looked up via JNDI. The functionality is essentially equivalent to lookups via the **IComponentsLookup** interface.

When looking up a component, it is typically required to specify the expected return type. When using JNDI URLs this can be accomplished via a **type** query parameter. For example, when looking up a JDBC data source (see Data Source Components) that is declared in a component repository as the component **mymodule/dataSource**, the call

```
IComponentsLookup.INSTANCE.lookup("mymodule/
dataSource",javax.sql.DataSource.class);
```

is equivalent to

```
new InitialContext().lookup("components:mymodule/dataSource?
type=javax.sql.DataSource");
```

and both calls return a (shared) data source instance.

3.2.10. Link Components and Component Linking

At times it can be useful to make an existing component definition available under another module and component name. This can be achieved using link components.

An important example is the construction environments from base environments as implemented by the sample applications. By convention the environment module contains actual runtime configuration for the web server, database datasources, user realms and in particular system states and worker process configurations. If we want to use different subsets of these configurations, e.g. because we define different systems based on the same basic distribution, we could copy and adapt the environment. That may however require to keep copies of complex configuration sets that would need to be updated once the underlying, shared, implementation changes. To mitigate that components may be linked similarly to symbolic links in Linux file systems.

In the case of the sample systems, components from the z2-base.base **environment_base** module are combined into a custom **environment** module by linking those required and overwriting those

modified. This saves duplication of the web server and worker configurations for example, while we keep the home layout configuration.

In order to declare a link component, define a component of type **com.zfabrik.link** and supply a target component using the **link.targetComponent** property. For example, re-using a web server configuration from the environment_base module would look like this:

```
#
# link to environment.base for defaulting
#
com.zfabrik.component.type=com.zfabrik.link
link.targetComponent=environment.base/webServer
```

Link components are built-in with z2's component handling and operations are delegated to declared target components:

- Lookup as implementation of an interface or class via e.g. **IResourceHandle.as(…)** or **IComponentsLookup.lookup(…)** in general (see the exception below).
- Component resource access via e.g. **IComponentsManager.retrieve(…)**.

Some operations are not delegated however:

- Lookup as **IResourceHandle** or **IResourceObserver**
- Component Descriptor retrieval via **IComponentsManager.get()**
- Search of components via **IComponentsManager.findComponents(…)**.

This leads to subtle but meaningful and important differences in behavior when working with linked components vs. working with ordinary components. For example, in the case of a linked component, a lookup for **IComponentDescriptor** would result in the descriptor of the target component, while a call to I**ComponentsManager.get()** for the same component would return the descriptor of the link component.

A target component may itself be a link component. Where link resolution is implemented, it is done indefinitely until reaching an ordinary component.

3.3. Unit of Work and transaction management

The z2-Environment does not mandate any specific way of implementing transaction management.

It does however have a concept of a *unit of work* that is used by parts of its implementation and that is the underpinning of the simple, but rather useful, built-in Java Transaction API (JTA) implementation.

A unit of work is a well-defined part of the control flow on one thread of execution that resources such as database connections can bind to and learn about whether all work should be committed or rolled back at the end of it. The WorkUnit API that is part of the Z2 core APIs implements this abstraction.

All threads managed by the z2-Environment wrap their work using this API and when extending the

z2-Environment with custom threading implementations, it is suggested that you wrap the actual work using the WorkUnit API, so that at least the z2 infrastructure can integrate cleanly and optimize resource usage.

The JTA implementation provided in the module **com.zfabrik.jta** provides a standard UserTransaction implementation that integrates with the WorkUnit API and thereby provides a robust transaction management abstraction that greatly simplifies integration with persistence tools like Hibernate JPA.

It can be looked up using the global JNDI name

```
components:com.zfabrik.jta/userTransaction
```

Note that **com.zfabrik.jta** is not a full-blown transaction manager that supports distributed transactions and corresponding protocols. It is fine for typical non-distributed transaction situations however.

In conjunction with the z2 provided database connection pooling (see ZFabrikPoolingDataSource) it is important to note that, if you choose work unit enlisting, then the WorkUnit abstraction defines transaction boundaries, so that automatically all database connections are enlisted with the current unit of work and committed or rolled back under control of the WorkUnit implementation.

In terms of the JTA implementation, this behaves as if there is already a transaction open on the thread.

The WorkUnit API supports nesting and suspending of units of work. With the JTA implementation this corresponds to nested and isolated transactions.

Please visit the Wiki page on transaction handling in Z2 to learn more about alternatives and how to integrate with a full-fletched transaction manager.

4. Constructing a Z2 system

This section describes how Z2 systems are assembled from repositories and how to construct your own system from what is provided on z2-Environment.net and your own repositories.

From a Z2 Core perspective it all starts with the Local Repository that is part of the core. In there, we define at least the Base Repository. The Base Repository typically points to some remote Git or Subversion based repository. Once Z2 has registered that repository, other repository may have appeared that will be registered as well which may lead to the appearance of yet more repositories and so on. Hence, in effect, we have a chain or tree of repositories with the Local Repository on its root as far as repositories contain declarations of repositories.

On the other hand, repositories have a priority (more on that below) that determine what repository has the right to say what a module contains.

Based on that mechanism you can construct a system definitions that consist of as few as one repository (if we do not count the core) or many repositories of which some are even shared between systems.

Before moving forward on that, let's have a look at the add-ons.

4.1. Add-ons

Add-ons add more functionality to Z2. Generally speaking, an add-on is a regular Git or Subversion

repository that holds one or more modules and is incorporated into a z2-Environment defined system via a component repository declaration (see Components and Component Repositories).

In other words, technically there is nothing particular about add-ons. It is the way they are used that is noteworthy. The idea is that you can pick the add-ons you need and add them on top of z2-base. Previously Z2 was available in distributions. Now you take z2-base and add what you need on top.

Add-ons provided on z2-environment.net are versioned just like z2-base, so that there is no complicated version vector. Also add-ons have some documentation in the z2-Environment Wiki and come with some samples.

5. Developing with the z2-Environment

So far we have learned about the principles behind the z2-Environment and how to configure and run it. This section is devoted to the development using Z2.

In principle you would not need any tool support. You could simply check out files from your favorite repository, use some text editor or your favorite integrated development environment to add projects and files or to modify files as you wish, commit your changes and synchronize the runtime that would do whatever else is needed.

While that is good news already, there are some simple tools that make your life still easier and give you a development experience you have probably not experienced before in Java environments.

The whole approach to local development using the z2-Environment is currently based on two tools:

a) The *Development Repository* – a component repository implementation that allows you to selectively and quickly test modifications

b) The *Eclipsoid Eclipse plug-in.* A plug-in to the popular Eclipse development environment that resolves project dependencies from a running z2-Environment.

5.1. A note on what JDK to use

Different Z2 versions support or require different Java versions.

Z2 Version	Required Java Version	Supported Java Version
< 2.4	6	6,7
2.4, 2.5	8	8
2.6	9	9,10, (11 not known)

As Z2 compiles Java code, it has to make a decision on what language level to compile for. Typically however you do not need to worry about this, as by default, Z2 simply sticks to the version of the Java Development Kit (or Java Runtime Environment) it is currently executed with.

That is, if you run a Java 8 JDK, then Z2 will compile for Java 8.

You can however enforce a language level to compile with using the system property **com.zfabrik.java.level**. Valid values are “6”,”7”, “8”, “9”, “10” for the respective Java version.

5.2. Workspace development using the Dev Repository

The Development Repository (or short Dev Repo), works by checking a file system folder for project sub folders that contain a file called **LOCAL** and scans for components inside.

It expects to find a component repository structure as detailed in Components and Component Repositories.

The Dev Repo has a high priority within the chain of component repositories. That means that whatever it finds, it will typically win against definitions provided from other component repositories.

By default, the Dev Repo is configured to look for changes in sub folders (three levels deep by default - but see below) of the folder that contains the core installation. That is the reason behind the folder structure described in the next section:

When using subversion and checking out the Z2 core into your Eclipse workspace the Dev Repo will find your projects. When using Git and importing projects from a working tree of a repository that is next to the Z2 core, the Dev Repo will find your projects as well.

This is how it all ties together: Given the Dev Repo is able to find your project (that means a module to z2 if accepted), you simply put file called **LOCAL** into the project's root folder and the project and all its components will be picked up with preference by the Dev Repo next time you trigger a synchronization.

That sounded a little complex, but as you will see next, together with the Eclipsoid tool it all rounds up nicely.

Before going there it is noteworthy that the Development Component Repository has use cases beyond development. Sometimes it handy to override centrally defined components, for example to modify web server ports or data source configurations, via the Dev Repo.

By modifying the system property **com.zfabrik.dev.local.workspace** you can influence where the Dev Repo scans for armed modules.

Using the system property **com.zfabrik.dev.local.depth** you can influence how *deep* the Dev Repo scans for LOCAL files. Depth is measured in path distance from the roots set via the system property **com.zfabrik.dev.local.workspace**. This setting defaults to a value of 3, which means that

```
folder1/folder2/folder3/LOCAL
```

would be found, which would then make up for module **folder3**.

5.2.1. Recommended folder structure

Using Git or Subversion makes no difference in the non-development folder layout of a Z2 installation. In development however there is a small but noticeable difference.

In a Subversion setup, the folder that holds the Z2 core check out is also used as development workspace. That is, you will have a workspace folder, say **workspace** and in that workspace the Z2 core checkout as well as other projects, typically corresponding to Z2 modules as far as Z2 is concerned. E.g.:

```
workspace/z2-base.core
```

```
workspace/com.acme.some.project
...
```

In a Git setup, the development workspace folder is a folder next to the clone of the Z2 core repository. Assuming you installed into **install** and the workspace folder is called **workspace** your structure would look like this:

```
install/z2-base.core
install/some.other.repo.clone
install/workspace
...
```

Note: In both cases, the search path for "armed" project is the same for the Development Repository (see above).

5.3. The Eclipsoid Plug-in

The Eclipsoid plug-in for the Eclipse development environment comes with the z2-base system and can be installed from the local update site at

http://localhost:8080/eclipsoid/update/site.xml

Alternatively, you can install it from the z2 environment server at

http://www.z2-environment.net/eclipsoid/update/site.xml

This plug-in provides a number of useful utilities for working with Z2. The most important functions are:

1. Trigger synchronization of the running z2-Environment from the IDE (*Sync*)
2. Download of dependencies as .jar files from a running z2-Environment (*Resolve*)

The Eclipsoid fixes an important problem in development of larger software systems in IDEs like Eclipse: Larger software systems consist of many different projects that have compilation dependencies between each other. That is, Java code in one project may not compile without having access to Java types defined in another project.

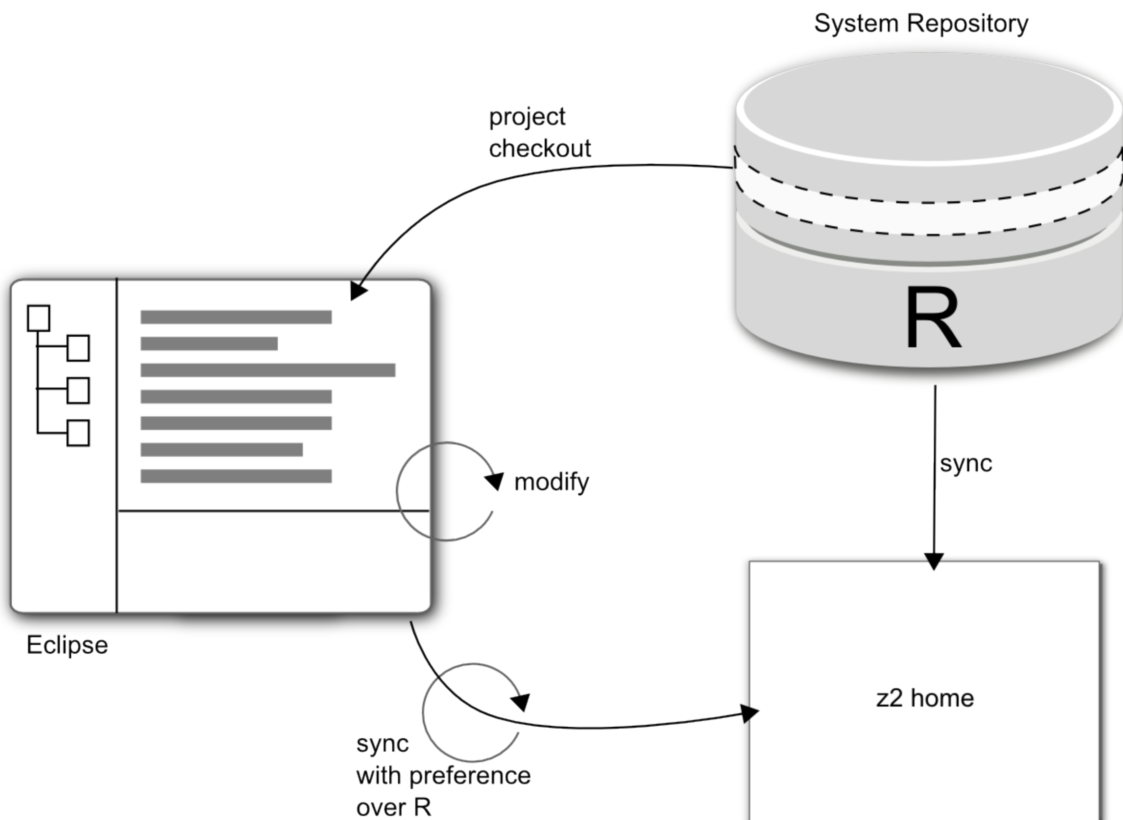
IDE's like Eclipse support local compilation of the code you are working on and show compilation problems early on. To do so however, the project's dependencies need to be resolvable. That is exactly what the Eclipsoid does: Upon *Resolve*, any Eclipse Java project that is recognized as a Z2 project will be introspected for Java components (see Java Components), and if found, their references will be resolved from the server-side and required Java definitions will be downloaded and provided to the project.

Technically, a Z2 project is any Java project that has the Z2 Class path Container (called "ZFabrik.Eclipsoid") in its class path (i.e. the ".classpath" file). You do not need to fix that by yourself though. Either by creating a project as Z2 project or by "Transform into Z2-project" you can let the plug-in do that for you.

That means: In order to work on a single project, from a possibly large solution, you check out that single project, invoke *Resolve*, and, from there on, modify and *Sync* repeatedly to test your modifications. When you are done you commit your changes and disarm projects again to make

sure the integrated content is effective.

The last step is actually somewhat depending on your setup. If your Z2 is hooked up with remote Git repositories, you may need to push changes to remote before.



Sync and *Resolve* can be invoked by pressing Alt+Y or Alt+R respectively or by clicking on the Z2 toolbar buttons.

Finally, the Eclipsoid can *arm* and *disarm* projects: Arming a project means to put an empty **LOCAL** file into it, and disarming means to remove that file again.

See the previous section for more details on the Dev Repo. Armed projects are shown with a green halo around the Z decoration in the project view.

5.4. Using the Offline Mode

One principle feature of Z2 is to provide for early integration across a team of developers using a central repository containing the latest changes – except for those modules you are currently working on by virtue of the development repository.

At times this can get in the way of productivity though. In particular when you find yourself in a situation of unreliable network connectivity.

For those cases you can turn Z2 into an offline mode during which component repository implementations will not attempt to fetch updates from non-local sources.

5.4.1. How the offline mode is used.

The offline mode is enabled or disabled by setting the system property **com.zfabrik.offline** to **"true"** or **"false"** respectively. The default value of this property is "false".

This works regardlessly of how this system property is set in the home process. Worker processes receive an updated value upon synchronization.

5.4.2. Enabling Offline mode at start time

Using the file **<Z2 home>/bin/runtime.properties** the offline mode can be enabled at starting time already. Note however that typically some collected offline content needs to be retrieved before working with a z2 system makes sense during development.

5.4.3. Toggling Offline Mode during Development

Alternatively the offline mode may be switched on or off using a check box on the Z2 GUI.

5.5. In-container unit testing using Z2 Unit

The z2Unit feature, integrated with z2-base, allows to run *in-container* tests in Z2 from anywhere where you can run JUnit tests. To learn more about the JUnit testing framework, please visit www.junit.org.

In-container tests are ordinary unit tests that run within the server environment. Standard JUnit tests run in an environment that often has little to do with the tested code's “native” environment, other than seeing the same types. Anything else, data base connectivity, domain test data, component and naming abstractions (e. g. JNDI) need to be firstly abstracted out from the tested code and secondly “mocked”, that is simulated one way or the other.

In small systems and for tests that have little environment dependencies that is well-manageable. In larger scenarios and for higher-level components this becomes unreasonable and assuring the correctness of the mocked up environment becomes a testing issue on its own.

The foundation of the z2Unit feature is a JUnit Test Runner implementation that delegates the actual test execution to a z2 server runtime. That is, although JUnit believes to run a locally defined test class, the actual test runs on a remote server running the methods and reporting results corresponding to the structure of the local test implementation (which indeeds matches its server-side equivalent).

Please visit the How_to_z2Unit Wiki page on z2Unit to learn how to practically use z2Unit.

If you want to automate out-of-container tests and cannot rely on the Eclipsoid to have a suitable class path, you should use the **com.zfabrik.dev.util/jarRetriever** tool to retrieve all required dependencies as described next.

5.6. Retrieving jars from Z2

In most everyday operations you do not need to think about binary build results when using the Z2 environment. Sometimes however, in particular when running or inspecting code outside of Z2 you it is required to have compiled binaries at hand.

Using the **com.zfabrik.dev.util/jarRetriever** tool you can request binaries of a set of Java components including dependencies. This tool is an example of a Main program running using an embedded Z2 environment. That is, in order to run it you do not need a Z2 server running. You do however need a Z2 home installation.

See JarRetriever for more info on the jarRetriever. Also see Embedded runtime and the Main Runner for more info on the embedded mode and the MainRunner.

One particular use case is to retrieve Jars from Z2 within an ANT script, for example to automate unit testing.

The following snippet is an example on how to retrieve all jars, including dependencies for some components from Z2:

```
<!-- fetch all libs →
<java classpath="${z2home}/bin/z_embedded.jar"
   classname="com.zfabrik.launch.MainRunner" fork="true">
   <!-- general config →
   <sysproperty key="java.util.logging.config.file"
          value="logging.properties" />
   <sysproperty key="com.zfabrik.home" value="${z2home}" />
   <sysproperty key="com.zfabrik.mode" value="development" />
   <sysproperty key="componentName"
          value="com.zfabrik.dev.util/jarRetriever" />
   <!-- output folder →
   <arg line="-out ${output}" />
   <!-- project to retrieve binaries from →
   <arg line="${components}" />
</java>
```

In this example the following properties are expected:

${output}	The folder to store the retrieved jar files
${components}	A blank-separated list of components to retrieve the jars from
${z2home}	The installation folder of the Z2 home that is being used to load the jar files from

6. *Enhanced Basis Features*

6.1. The Binary Hub Component Repository

In some cases, it is not desired to have the Z2 runtime access source code repositories directly, for example so that no source code is ever stored on production machines. Other reasons may be to remove the load of compilation from production nodes.

The Hub Component repository addresses this problem by providing the following pieces:

- A providing side, that serves all modules and component available to the system in a pre-compiled form (as far as compilable code is involved)

- A client side that connects to the providing side

So, instead of connecting to an original source of components, the Hub Component Repository enables an operational approach where some Z2 runtimes see all system content in pre-compiled

form only.

Please read on at How to use the Hub Component Repository.

6.2. The Gateway for Zero-Downtime-Upgrades

The Gateway module implements a "zero-downtime-upgrade" feature in Z2. Specifically, it uses the worker process management of Z2 in conjunction with an intermediate reverse proxy style Web handler to implement the following feature:

Upgrading a stateful Web application, i. e. a Web application that stores user data in its HTTP session typically implies downtime, and if the session state is not serializable and persisted during the upgrade, it does additionally imply that user state gets lost and typically that users need to log on again.

Using the Gateway, running sessions may be preserved and worker resources may still be assigned on the current software revision for as long as there are running sessions during a node upgrade and until all sessions have been terminated. The typical application of this feature is to roll out functional and user interface corrections without interrupting users. Users can switch over to post-upgrade software by terminating their session (e. g. via a log out) and starting a new one (e. g. by logging in again).

The approach behind the Gateway feature is simple:

1. Allow separation of user sessions across worker processes
2. Provide an entry point to Web Applications that is capable of identifying what worker process is serving an associated session and of routing to that worker process.
3. Enhance worker process management with the capability of identifying stale worker processes that will not serve any user request in the future.

Please read on at How to Gateway to learn more about this feature.

7. Component type reference

This section describes the component type available in a z2-base system. Add-ons may provided additional component types. For those, please visit the Z2 Wiki at

http://redmine.z2-environment.net/projects/z2-environment/wiki.

7.1. Core component properties

Components in a Z2 component repository are declared using a set of properties, name-value pairs, that state the essential characteristics (beyond the name) of a component.

Typically, the component type (also a property, see below) defines the set of properties that make sense declaring. Some components however look for declarations in other components. As an example visit the System State component type below.

Very few properties are built-in with the z2 core and apply to any component:

name	values
com.zfabrik.component.type	The type of the component. The value of this property determines the Component Factory that implements the

name	values
	semantics of the component.
com.zfabrik.component.dependencies	A comma-separated list of component names. Components listed should implement **IDependencyComponent.** That interface will be invoked before providing from the declaring component and the declaring component will depend on all listed components.

Component dependencies allow to make sure that other components may be “prepared” before some particular component becomes used. This can be handy when some functionality of your solution depends on a side-effect established by another component. For example a web application may depend on a successful database migration check or another web application.

In order to preprepared a component implementation needs to provide implementations IDependencyComponent, as for example Web Apps do.

Note that other component types, such as System States may define properties that apply to yet other components.

7.2. “Any” components (core)

Any components may represent, as the name tries to indicate any sort of interface or aspect. In short, implementations of any components simply extend the Resource Management resource base class Resource (JAVADOC) pointer.

Typically, “any” components are only useful, if you need to satisfy some generic interfaces like **IDependencyComponent** but there is no more narrowly defined semantic provided in the form of a component factory.

That said, unless you have a problem that may demand an “any” component, you do not need to worry about them.

Properties of an “Any” Component:

name	values
com.zfabrik.component.type	**com.zfabrik.any**
component.className	Name of the class that implements **com.zfabrik.resources.provider.Resource**

7.3. Component Factories (core)

See Components and Component Repositories for details on the concept of component factories.

In general a component factory implementation is an implementation of the interface **com.zfabrik.components.provider.IComponentFactory**. When called, it is asked to return an extension of **com.zfabrik.resources.provider.Resource** that represents all runtime aspects of the component of the passed-on name.

As a short cut, the class name given by the property **component.className** in the component's descriptor may name a class that extends **com.zfabrik.resources.provider.Resource** rather than

implementing the factory interface above.

In that case, the extension class must have a constructor that takes a single String parameter and it will be instantiated for a given component by its name when required (i. e. when otherwise the factory interface would have been called.

Only one component factory per type name may be declared.

Properties of a Component Factory Component:

name	values
com.zfabrik.component.type	**com.zfabrik.componentFactory**
component.className	Name of a class that implements **com.zfabrik.components.provider.IComponentFactory** or name of a class that extends **com.zfabrik.resources.provider.Resource.** See also above.
componentFactory.type	Name of the component type implemented by this factory. Components that declare to be of this type are managed by resources provided by this type.

7.4. Data Source components (z2-base)

Data source components allow to manage JDBC data sources as z2 components. When present, the built-in support for JNDI lookups (see Java Naming and Directory Interface (JNDI) support of the z2-Environment can be used make these datasources accessible in a standard way for widely used Java frameworks such as Java persistence providers, or they may be used directly.

The benefits of specifying JDBC data sources as z2 Components lies in the simple maintenance of their configuration. You are in no way limited to using this component type, when you need a datasource. At times it may be more suitable to leave your data source configuration for example in a Spring application context and expose it as a bean to make it re-usable across modules.

Data source configuration is split into two parts: General configuration and Data Source implementation specific configuration.

General Properties of a Datasource Component:

name	values
com.zfabrik.component.type	**javax.sql.DataSource**
ds.type	The type of data source used. Supported values are **NativeDataSource** or **ZFabrikPoolingDataSource**. See below.
ds.enlist	The data source may be enlisted with the WorkUnit. The WorkUnit API provides a simple way to attach shared resources on the current thread of execution for the time of a unit of work (typically a web request, some batch job execution) as implied by thread usage (see ApplicationThreadPool). Supported values are **none** and **workUnit.** Default value is **workUnit.**
ds.dataSourceClass	If set, the specified data source class will be loaded as data source implementation using the private class loader of the Java module of the component holding the data source definition.

name	values
	When specifying this class in conjunction with the using **ZFabrikPoolingDataSource** as type, configuration properties will be applied to both and the pool will request new connections from the specified data source. Alternatively, the pool may be configured to use a driver class. See below.

See also DataSourceResource.

7.4.1. Data Source Specific Configuration

When specifying a data sourc class but also when using the built-in pooling data source, properties of the data source implementation class can be specified as Java Beans properties using the syntax below:

ds.propType.<prop name>	Type of the property. Can be **int**, **string**, or **boolean**. Default value is string.
ds.prop.<prop name>	Value of the data source property to be set according to its type setting above.

7.4.2. Data Source Types

Currently the Data Source support allows to specify two different types of data sources:

NativeDataSource

When declaring a native data source, the ds.dataSourceClass must be specified to name a data source implementation class.

All further configuration of the data source is done generically using the property scheme below,

ZFabrikPoolingDataSource

When declaring a ZFabrikPoolingDataSource a z2 provided data base connection pool implementation will be used that has the following configuration properties:

Name	Type	Value
driverClass	string	Name of the actual JDBC Driver implementation class. E.g. **com.mysql.jdbc.Driver** for MySQL.
url	string	JDBC connection url
user	string	User name for authentication at the data base.
password	string	Password for authentication at the data base.
maxInUseConnections	int	Maximal number of connections handed out by this pool. This number may be used to limit database concurrency for applications. Requesting threads are forced to wait for freed connections if this limit has been exhausted. Make sure threads are not synchronized on shared resources when requesting connections and when this limit is less than your theoretical application concurrency as this may lead to thread starvation.

maxSpareConnections	int	Number of connection held although not currently used by the applications.
connectionExpiration	int	Connections will be closed after this number of milliseconds has expired since creation and when returned to the pool. This setting can be used to make sure stale connections get evicted although not detected otherwise by the pool.
connectionMaxUse	int	Connections will be closed after this number of times they have been handed out from the pool and when returned to the pool. This setting can be used to make sure connections only serve a limited number of requests.

See also PoolingDataSource.

A typical ZFabrikPoolingDataSource configuration looks like this:

```
#
# MySQL driver configuration
#
ds.dataSourceClass=com.mysql.jdbc.jdbc2.optional.MysqlDataSource
ds.prop.user=z2
ds.prop.password=z2
ds.prop.url=jdbc:mysql://localhost:3306/z2_samples?autoReconnect=true
#
# Generic pooling config
#
ds.prop.maxInUseConnections=10
ds.propType.maxInUseConnections=int
ds.prop.maxSpareConnections=5
ds.propType.maxSpareConnections=int
ds.prop.connectionExpiration=60000
ds.propType.connectionExpiration=int
```

7.5. File system component repositories (core)

File system based repositories are the most straightforward repositories. All that is required is a file system folder that holds components and component resources in a structure as described in Components and Component Repositories. As always for component repositories, it is important to make sure they are started early on in the life-cylce of a z2 runtime.

Note that unlike the development repository (see Developing with the z2-Environment), the file system repository is not robust under modifications: Resources in the folder structure of the file system repository will be accessed at any time the z2 runtime requires to – which may be significantly later than the latest synchronization that decided about invalidations due to changes. Resources should not be modified in the meantime to assure consistency.

Properties of a File System Component Repository Component:

name	values
com.zfabrik.component.type	**com.zfabrik.fscr**

name	values
fscr.folder	Store folder, i.e. the file system folder that holds the actual resources of the repository.
fscr.base	If the value of **fscr.folder** is not an absolute file system path but a relative path instead, this setting specifies what it is evaluated in relation to. Allowed values are "**home**" (default) and "**here**". If set to "**home**", the basis of evaluation is the z2 Home folder (see #FolderStructureOfHome). If set to "**here**", the basis of evaluation is the resource folder of the declaring component. This is useful, if you want to declare FS component repositories from FS component repositories in a relative fashion.
fscr.checkDepth	Component folder traversal depth when determining the latest time stamp. Set to less than zero for infinite depth. Default is -1.
fscr.priority	Component repository priority. See **IComponentsRepository**. Default is 250.

7.6. GIT component repositories (core)

When using a Git based component repository, the z2 runtime manages a local clone of another Git repository. This allows to declare Git component repositories that refer to a local Git repository, typically present in a development setup, or to remote Git repositories, used for production setups.

In both cases, when synchronizing, the component repository will pull updates from the configured repository and check for modifications by inspecting the local workspace, i.e. the Git workspace maintained by the z2 environment runtime itself.

Note: When specifying a local repository in gitcr.uri the relevant branch is still the one configured in the component properties (see below), not the checked out branch of the local repository.

By default, the root of the repository is considered the root of the component repository as well. That is, modules would be identified as direct sub folder of the Git repository root path.

A Git component repository may be configured to use other locations inside the repository instead using the **gitcr.roots** configuration property. Instead of checking for modules at the root of the Git repository, modules will be searched at the given relative paths in the repository. For example, given **gitcr=repo1,other/repo2** the relatives paths **repo1** and **other/repo2** will be searched.

The order of paths in the comma-separated list determines the priority in overlaying in decreasing order. That is, using the example above, if a module **myModule** would be found at **repo1/myModule** as well as in **other/repo2/myModule**, the latter definition would be ignored.

See also Git support for more information on Git support.

Properties of a Git Component Repository Component:

name	values
com.zfabrik.component.type	**com.zfabrik.gitcr**

name	values
gitcr.uri	The URI to the Git repository to clone from. This can be an absolute path, a local path relative to .../bin, or a remote URL. See the Git documentation for examples.
gitcr.priority	The priority of the repository. Defaults to 500.
gitcr.ref	A ref to check out as active content. This can be • a tag (**refs/tags/<tag name>**), • a commit (some SHA, e.g. **a9c96be455a40b35ee3cd33c8de07b4de87a6240**), • or a remote (**refs/remotes/origin/<branch>** or shortened to **origin/<branch>**). In practice this can be anything used for the JGit check out command (omitting the branch creation part) which is very similar to what may be used in the git checkout command (omitting anything beyond checkout, such as branch creation). Use this option, if you need more control then given by **gitcr.branch**). Note that the latter will be used with preference and hence needs to be omitted, if you want to use **gitcr.ref**.
gitcr.branch	The branch to checkout. Setting this is equivalent to setting a **gitcr.ref** value **refs/remotes/origin/<branch name>** and will be considered with preference over **gitcr.ref**.
gitcr.roots	A comma-separated list of repository local paths that serve as roots of the actual component repository content. The order is in decreasing priority. See also above. Defaults to the empty path which identifies the root of the repository.

7.7. Home Layouts

Home layouts define a set of worker processes to run. Home layouts are one of the few components that only run on the home process when operating the z2-Environment in server mode.

See Understanding a Z2 Home for more details the home process and worker processes.

You can use home layouts to define a static OS process layout of all z2 home runtimes of your system as well as you can use home layouts to have heterogeneous cluster layouts, that is, a setup where many z2 home installations share one system definition but run different sets of worker process configurations.

At one point in time, a home process will only maintain one home layout. To specify the home layout to use, use the system property com.zfabrik.home.layout and set it to the name of the particular home layout component – typically in a mode line of the **launch.properties** file (as described in Folder Structure of a Z2 Home)

When the Home Layout component is loaded, it will try to load the worker processes specified and depend on them subsequently.

Properties of a Home Layout Component:

name	values
com.zfabrik.component.type	**com.zfabrik.homeLayout**
home.workers	A comma-separated list of worker process components. See below.

A typical home layout declaration looks like this:

```
com.zfabrik.component.type=com.zfabrik.homeLayout
home.workers=\
   environment/webWorker,\
   environment/jobWorker
```

7.8. Java components (core)

Java components are among the very few very essential component types already defined in the Z2 core. The knowledge about Java components is essential to the environment so it can bootstrap.

The Java component implementation takes care of the following tasks:

- Resolving includes for assembly
- Resolving references of Java components for class path computation and class loader management
- Triggering (re-) compilation of source code in a Java component using the compiler API

The mechanisms around references and includes between Java components and the separation of Java components into public (api), private (impl), and test, are the underpinnings of the software modularization features of z2, which is why we discuss these in some depth here.

7.8.1. Classloaders

The class loader concept of the Java platform provides a powerful name spacing mechanism on the type system. While in the beginning that seems to be of little concern, in more complex scenarios isolation within the type system in conjunction of sharing of types between modules of a solution becomes the catalyst of successful modularization.

Isolation means that modules on the platform may use types without sharing them, that is without making them visible to other modules. That can be important for various reasons:

- Implementation types should be hidden from potential users so that modifications do not break using modules.(encapsulation).
- In particular third party libraries used in the implementation of a module may conflict with other versions of similar libraries so that exposing them would lead to unnecessary risks on the consuming side (multiple versions)

Sharing of types on the other hand allows to refer to the very same types from different modules and, as they are shared, provides an efficient type safe way of communicating state between modules:

- By publishing an API, modules may expose services efficiently to other modules

Based on these mechanisms, modularization for Java components on Z2 provides the ability to maintain a system of named modules that have defined contracts among each other while still maintaining local integrity and cohesion.

The class loading system in Z2 is based on a ancestry-first, multi-ancestor scheme. Effectively, a Java component will have two class loaders at runtime. One for the API, one for the Implementation and both ask their ancestors (other class loaders) first before searching local resources.

The API class loader will have ancestors corresponding to all Java components identified by the public references. The implementation class loader will have the API class loader as ancestor and ancestors corresponding to all Java components identified by the private references of the Java component (see below).

7.8.2. Includes

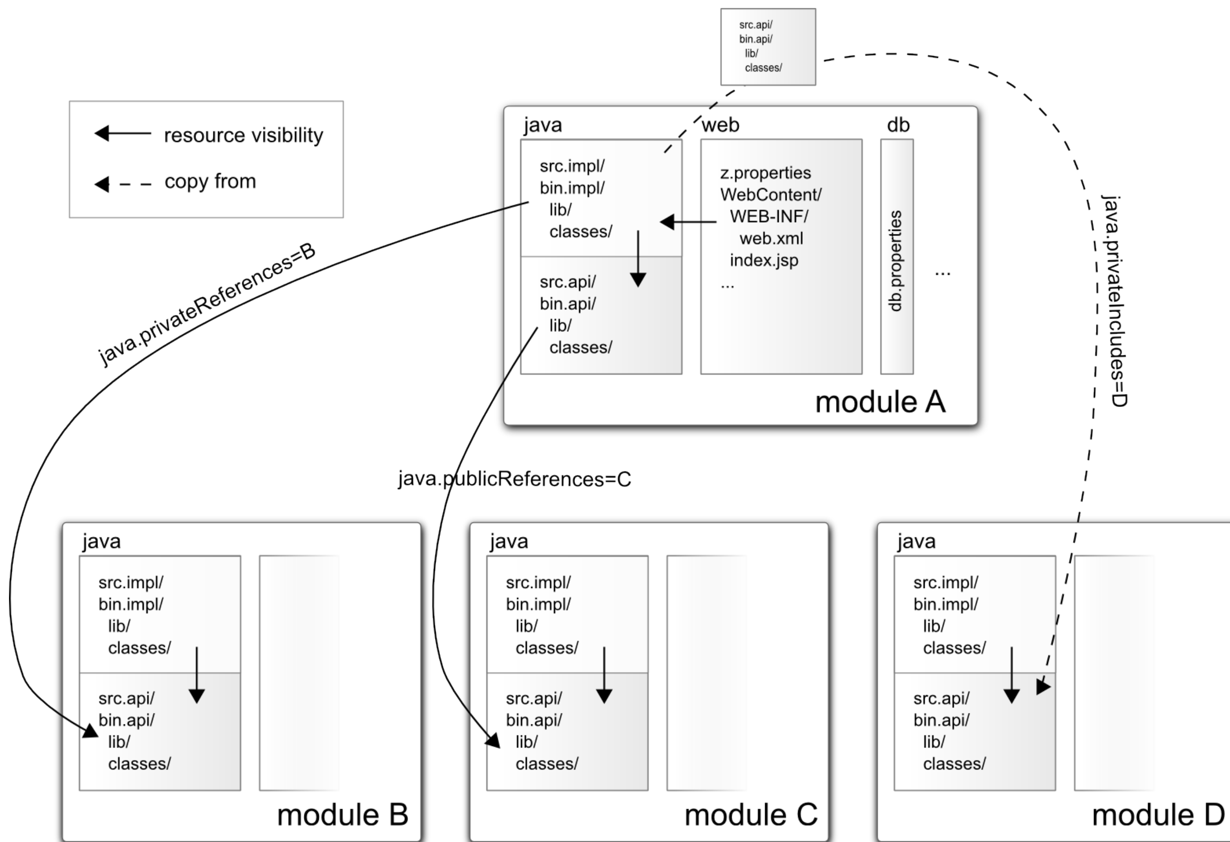
Another important mechanism supported by the Z2 Environment is so called "Java includes". The references feature described above allows sharing of types and class path resources without duplicating them at runtime.

There are cases however where duplication of types is necessary – although that is fortunately the exception:

- Frameworks like the Spring Framework supply pre-compiled libraries that contain "adapters" for various other frameworks that may not be present on the using application. The late linking qualities of the Java VM supports unresolvable type references as long as they are not needed. In this case, the library must be used in the class loading name space of the using application to make sure it gets appropriate type visibility.
- Some libraries attach information about the using application to the class loading namespace itself, e.g. via class variables. In that case, sharing types can easily lead to unpredictable behavior as state from different class loading name spaces may override each other.

The use of includes, actually in most cases only "private includes" implements exactly that. The Java resources of the included component get copied into the using Java component and hence are used as if provided by the using Java component.

The picture below shows a simplified example overview over the reference and the include mechanisms:



Properties of a Java Component:

name	values
com.zfabrik.component.type	**com.zfabrik.java**
java.publicReferences	Points to another java component whose public types will be shared with this one (and maybe others). Everything referenced as public reference will be visible to the public interface of the referencing component as well as to all referencing the referencing component. In other words: References are transitive. In particular, anything required to compile the public types of a Java component must be referenced via this reference property. Components may be specified as a comma-separated list. Component names that have no "/" will be defaulted by appending "/java".
java.publicIncludes	Points to **com.zfabrik.files** or **com.zfabrik.java** components that must have a **bin** (or alternatively a **bin.api,** for Java components) folder that will be included into this java component's public java resources. The component may also have a **src** (or alternatively **src.api**, for Java components) folder that will be copied before compilation into **src.api**.
java.publicClassPathInclusion	Custom regular expression for classpath computation of the public classloader. Only library files matching the specified pattern will be included into the respective classpath

name	values
	computation. This has no effect on the inclusion of .class files. Defaults to “**^.*(?<!-sources)\\.jar$**” (i.e. including all jar files except those ending on “-sources” before the file extension).
java.publicCompile.order	Like **java.compile.order** below, but only applying to the public resources and considered with preference.
java.privateReferences	Points to another java component whose public types will be shared with this one (and maybe others) Nothing referenced as private reference will be automatically exposed to the public interface of the referencing component nor to other components. Anything needed to compile the private types of a Java component, must be referenced as a public reference, be part of the public types of that component, or be referenced via this reference property. In other words: The private types automatically see the public types and transitively anything referenced publicly as described above. In addition, to use more types in the "private implementation section" of a Java component, types that will not be exposed to referencing components, use this reference property. Components may be specified as a comma-separated list. Component names that have no "/" will be defaulted by appending "/java".
java.privateIncludes	Points to **com.zfabrik.files** or **com.zfabrik.java** components that must have a **bin** (or alternatively a **bin.api**, for Java components) folder that will be included into this java component's private java resources. The component may also have a **src** (or alternatively **src.api**, for Java components) folder that will be copied before compilation into **src.impl.**
java.privateClassPathInclusion	Custom regular expression for classpath computation of the private classloader. Only library files matching the specified pattern will be included into the respective classpath computation. This has no effect on the inclusion of .class files. Defaults to “**^.*(?<!-sources)\\.jar$**” (i.e. including all jar files except those ending on “-sources” before the file extension).
java.privateCompile.order	Like **java.compile.order** below, but only applying to the private resources and considered with preference.
java.testReferences	Points to another java component whose public types will be shared with this one (and maybe others) if the execution mode, as defined by the system property (see **Foundation.MODE**} is set to **development**. Test references extend the private references. In conjunction with the tests source folder this allows to add test code and corresponding

name	values
	dependencies that will be ignored by the runtime unless running in development mode.
java.testIncludes	Points to **com.zfabrik.files** or **com.zfabrik.java** components that must have a **bin** (or alternatively a **bin.api,** for Java components) folder that will be included into this java component's test java resources. The component may also have a **src** (or alternatively **src.api,** for Java components) folder that will be copied before compilation into **src.test**.
java.testClassPathInclusion	Custom regular expression for classpath computation of the test classpath. Only library files matching the specified pattern will be included into the classpath computation. This has no effect on the inclusion of .class files. Defaults to "**^.*(?<!-sources)\\.jar$**" (i.e. including all jar files except those ending on "-sources" before the file extension).
java.testCompile.order	Like **java.compile.order** below, but only applying to the test resources and considered with preference.
java.compile.order	The compile order must be defined in java components that also contain non-java sources - e.g. scala. This property can be omitted for pure java components, otherwise one has to define all compilers in the right order - e.g: scala, java

7.9. JUL configurations (z2-base)

The standard logging implementation contained in the package java.util.logging (or JUL for short) of the Java SE distribution can be configured using components of type java.util.logging.

The z2 Environment implementation uses JUL throughout (rather than log4j or other logging mechanisms). Defining java.util.logging components provides an easy way to distribute log configurations without the need to modify command lines and without need to restart the runtime,

Components of type java.util.logging are expected to provide a file called **logging.properties** in their resources (see for example the component **environment/logging** in z2_base/base). That file will be applied using **LogManager.getLogManager().readConfiguration(...)** every time the component is prepared (as in **IDependencyComponent**, i.e. as part of a dependency resolution), e.g. when (re-) attaining a participated system state.

Properties of a JUL Configuration Component:

name	values
com.zfabrik.component.type	**java.util.logging**

7.10. Link Components

Link components can be used to re-use existing component declarations and resources by another module and component name. Details are explained in Link Components and Component Linking.

Link components are declard using the component type **com.zfabrik.link**.

Properties of a Link Component:

name	values
link.targetComponent	Name of the target component.

7.11. Log4J configurations (z2-base)

Components of type **org.apache.log4j.configuration** are handled exactly as components of type **java.util.logging** (see right above), except that a file called **log4j.properties** is expected and loaded using Log4J's **PropertyConfigurator** API (see the Log4J documentation for the specifics of Log4J configuration).

Properties of a Log4J Configuration Component:

name	values
com.zfabrik.component.type	**org.apache.log4j.configuration**

7.12. Main Program Components (core)

While the z2-environment has built-in support for long running worker processes it is perfectly usable to run command line programs via the Main Runner as described in section #embedded.

In order to implement a Main Program component to be called from the command line, the component implementation is required to either

- support lookup as **java.lang.Class** returning a class object that has a main method,
- or make use of the component type **com.zfabrik.mainProgram** and declare an implementation class that has a main method.

Note: A main method, is a method of the signature

```
public static void main(String[] args)
```

In the first case, the declared component type is irrelevant, as long as a lookup requirement is satisfied. For example an "any" component serving as a Main Program could be implemented like this:

```
public class MainResource extends Resource {

   @Override
   public <T> T as(Class<T> clz) {
         if (clz.equals(Class.class)) {
               return clz.cast(MainResource.class);
         }
         return super.as(clz);
   }

   public static void main(String[] args) {
         System.err.println("hello");
```

```
    }
}
```

In the second case, it is not required to implement a typed lookup implementation via the as(..) method. Instead a class with a main method suffices.

Example:

Component declaration:

```
com.zfabrik.component.type=com.zfabrik.mainProgram
component.className=test.Main
```

Main class implementation:

```
public class Main {
    public static void main(String[] args) {
          System.err.println("Hello Main");
    }
}
```

Assuming the component is declares as mymodule/main, the environment variable Z2_HOME is set, the command line

```
java -DcomponentName=mymodule/main -jar z_embedded.jar
```

will invoke the main program (note: for experimentation you might want to work with a locally defined development state using the dev repo. In that case set the system property **com.zfabrik.dev.local.workspace** on the command line accordingly - e.g. to the folder above your development modules - see also #devRepo).

All shared component properties, such as those declaring dependencies, are supported. A main program runs in exactly the same context as any other component defined. This make Main Program components an extremely useful command line interface of a potentially application system.

Properties of a Main Program Component:

name	values
com.zfabrik.component.type	**com.zfabrik.mainProgram**
component.className	Name of class that has a main method.

7.13. Maven component repositories (z2-base)

Maven component repositories allow to integrate artifacts from Maven artifact repositories without

copying them into your system. See Maven Repository Support for more details.

Properties of a Maven Component Repository Component:

name	values
com.zfabrik.component.type	**com.zfabrik.mvncr**
mvncr.settings	Specifies the location of the settings XML file relative to the components resources. This is expected to be a standard Maven configuration file. Defaults to settings.xml
mvncr.roots	A comman-separated list of root artifacts
mvncr.managed	Fixed artifact versions, if encountered during recursive root resolution. This corresponds to a <dependencyManagement> section in a Maven POM file.
mvncr.excluded	A comma separated list of artifacts that will be skipped during resolution of any root
mvncr.priority	Component repository priority. See **IComponentsRepository**. Default is 500.
mvncr.repository	Symbolic name of the repository. Use this to define a component name independent identifier for the repository that fragments (see below) can reference.

In order to deviate from the default resolution and mapping, maven repository roots may be specified with a query string syntax like this:

```
org.springframework.security:spring-security-aspects:3.2.2.RELEASE?versioned=true&scope=RUNTIME&excluded=commons-logging:commons-logging:1.1.2
```

Admissable query parameters are

param	meaning
com.zfabrik.component.type	**com.zfabrik.mvncr**
versioned	If set to true, the version part will not be removed from the java component name mapping and instead a versioned name is used. That is, in the case above, a java component **org.springframework.security:spring-security-aspects:3.2.2.RELEASE/java** would be mapped. This is useful if "non-default" versions are required.
scope	Any of RUNTIME, COMPILE, PROVIDED, SYSTEM, TEST. Corresponds to the corresponding Maven dependency scopes. If set, non-optional dependencies of the respective scope will be traversed to resolve dependencies.

param	meaning
excluded	Exclusions on the dependency graph.

7.14. Maven component repository fragments (z2-base)

A Maven component repository fragment allows to add dependency graph information to a Maven component repository (see above).

Properties of a Maven Component Repository Fragment Component:

name	values
com.zfabrik.component.type	**com.zfabrik.mvncr.fragment**
mvncr.component	The component name (or a comma-separated list of component names) of Maven component repository or Maven component repository fragment declarations this fragment adds to. DEPRECATED, use mvncr.repository instead.
mvncr.repository	The maven repository name (or a comma-separated list of repository names) of Maven component repository or Maven component repository fragment declarations this fragment adds to.
mvncr.roots	As above. Will be merged with the repo this fragment adds to.
mvncr.managed	As above. Will be merged with the repo this fragment adds to.
mvncr.excluded	As above. Will be merged with the repo this fragment adds to.

7.15. Subversion component repositories (core)

Subversion component repositories provide an easy and robust way to run z2 in a highly controlled and versioned yet scalable way.

See Components and Component Repositories for details on Subversion support.

Properties of a Subversion Component Repository Component:

name	values
com.zfabrik.component.type	**com.zfabrik.svncr**
svncr.url	URL of the subversion root folder of the repository. E.g. something like **svn://z2-environment.net/z2_base/trunk/base**
svncr.user	User name for Subversion authentication (optional).
svncr.password	Password for Subversion authentication (optional)
svncr.priority	Component repository priority. See **IComponentsRepository**. Default is 500.

By default, a Subversion component repository needs to be able to connect to the Subversion repository. In most cases this means that you need to be online when running a z2 environment,

even when developing.

In reality however, the repository has all required resources in its local caches and only checks for updates. In development situations it can be very handy to have the repository simply go by what is on the caches and continue developing in an Eclipse workspace gladly ignoring possible central modifications.

To enable that you can set the system property

```
-Dcom.zfabrik.repo.mode=relaxed
```

in **launch.properties** (see also #systemProps). In that case, failing to connect to a remote repository will be noted by a warning but otherwise ignored and the repository will try to satisfy component lookups from cached data.

To avoid problematic licenses, the z2 Environment does unfurtunately not come with complete built-in Subversion connectivity. Additional configuration steps are required once to complete subservion enabling on your side, as described in the Subversion How-To.

7.16. System States (core)

System states are abstract target configurations for z2 processes. Systems can easily develop into a non-trivial set of web applications, batch jobs, web service interfaces and more that interplay with each other to implement solution scenarios. Take for example an e-commerce web site: There is the actual shop front-end but also report generation, mass-emailing, shop content administration, etc.

It is handy to group components that form parts of an overall scenario and that need to be initialized beforehand, such as web applications, or update scheduling for analytical data aggregation.

System states support dependency declarations in two directions:

- Via the property **com.zfabrik.systemStates.dependencies** a comma-separated list of *components* that this state depends on and that must be *prepared* to have the state attained.
- Via the property **com.zfabrik.systemStates.participation** a comma-separated list of other *system state components* may be specified that are to depend on this system state and that cannot be attained unless this system state has been prepared (which equals attained in this case). This property may be declared on components of any type.

State dependencies are evaluated in an eager way. That is, even if some dependency fails to prepare, and eventually the system state will not be attain, all other dependencies will still try to be prepared.

System state dependencies are similar to component dependencies (see Core component properties). There are subtle differences however in that component dependencies are evaluated much earlier in the life cycle of a component than state dependencies. In essence that means, that the system state implementation may never evaluate participations if its component dependencies fail. Hence, on system states, it is preferable to not use component dependencies but rather the declarations above.

In practice, system states serve as a convenient, named place holder for parts of an overall scenario.

Attaining a system state means to “prepare” (see IDependencyComponent) all dependent component and do so again if one got invalidated and the system is to be attained again.

The system state feature is used by z2 in several places:

- All z2 processes (in server mode) have the target state **com.zfabrik.boot.main/process_up**

- The home process has a hard-coded target state **com.zfabrik.boot.main/home_up**
- Worker processes always have the target state **com.zfabrik.boot.main/worker_up**
- Component repositories should participate in **com.zfabrik.boot.main/sysrepo_up**
- Worker processes express their target configuration by a list of system states to attain (and keep attained). See below for worker process component configuration.

As mentioned above, in order to assign components as part of a system state, you can either list them as dependency components in the system state definition like this:

```
com.zfabrik.systemStates.dependencies=\
   mymodule1/comp1,\
   mymodule2/comp2
```

Or, in the case of other system states, mark them as participants of the system state. For example to your component's properties you could add

```
com.zfabrik.systemStates.participation=com.zfabrik.boot.main/worker_up
```

By convention, the most essential system states used and participated in for application components are declared in the "environment" module. We suggest to use:

- **environment/webWorkerUp** for web interfaces
- **environment/jobWorkerUp** for asynchronous data processing.

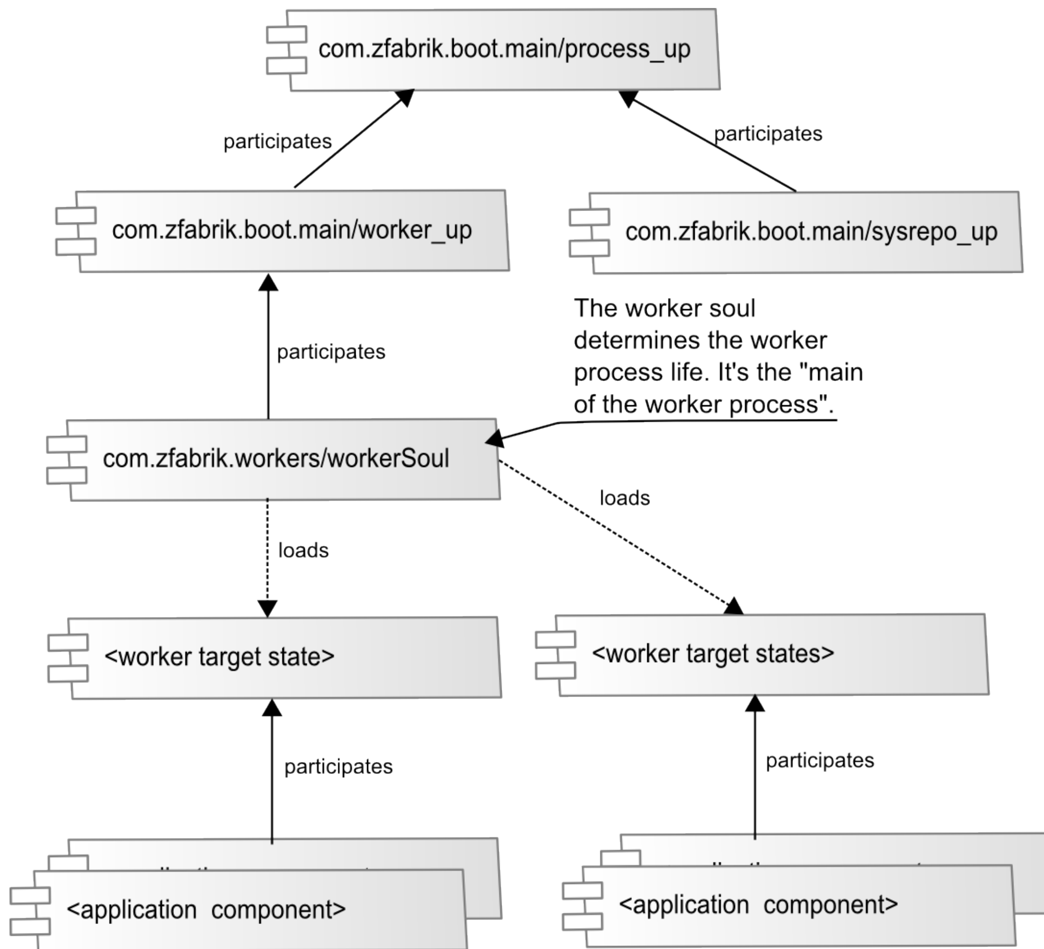
Larger scenarios may need more fine-grained scenario partitioning.

Declaring a system state requires no more that declaring a component of the corresponing type.

Properties of a System State Component:

name	values
com.zfabrik.component.type	**com.zfabrik.systemState**
com.zfabrik.systemStates.dependencies	A comma-separated list of *components* that this state depends on and that must be *prepared* to have the state attained.

The following diagram shows typical system state relationships in a worker process:



7.17. Web applications (z2-base)

Web application configuration is split into three parts:

1. The Java EE standard Web application deployment descriptor found at WEB-INF/web.xml the Web application's Web resources.
2. Container-specific extended Web application configuration. In the case of Jetty (currently the default Web container in z2) please check out the documentation on Jetty's WEB-INF/jetty-web.xml configuration file.
3. The z2 component descriptor that covers the remaining parts (such as the Web app's context path) and life cycle control inherent to z2.

Web applications have the following module structure in z2:

WebContent	Folder holding the standard Java Web application structure, such as the WEB-INF folder.
z.properties	Component descriptor of the Web application

Properties of a Web Application Component:

name	values
com.zfabrik.component.type	**com.zfabrik.ee.webapp**
webapp.path	Context path of the web application.
webapp.server	Component name of the web server to host this Web application.
webapp.requiredPaths	A comma-separated list of context paths this Web application relies on. This is an alternative way of defining a component dependency by Web app context path rather than by component name.
webapp.ContainerIncludeJarPattern	The jars that qualify for meta-data parsing can be limited in Jetty. In z2 the default is defined by the regular expression "(?!.+-sources\\.).*\\.jar$" (as Java string) which is all Jars except the source code jars (by convention). See also the Jetty Documentation.

7.18. Web servers (z2-base)

In order to run Web Applications, arguably the most prominent reason to run an application server, z2 integrates the Jetty Web Container.

There is no particular reason other than that Jetty is a well-embeddable, well performing, standard compliant web container. Based on z2's extensible component model, the way Jetty has been integrated, Tomcat could be integrated as well. Contact us, if that is important for you.

The component type **com.zfabrik.ee.webcontainer.jetty** configures instances of Jetty web servers. While in most cases you will not operate more than one, the component still provides the place to hold Jetty configuration. As an example have a look at the **environment/webServer** component in z2_base/base.

Properties of a Web Server Component:

name	values
com.zfabrik.component.type	**com.zfabrik.ee.webcontainer.jetty**
jetty.config	Names a Jetty configuration file (typically called jetty.xml) relative to the component's resource folder
jetty.override-web.xml	Names a override web.xml file, that can be used to override web application configurations for all web applications. The file name is considered relative to the component's resource folder
jetty.default-web.xml	Names a default web.xml file, that defines web application defaults for all web applications. The file name is considered relative to the component's resource folder

7.19. Worker Processes (z2-base)

Worker processes are managed by the home process when running in server mode. Worker processes improve robustness of the z2 runtime as applications running in some worker process do not impact applications running in another worker process nor and in particular, do crashing worker processes impact the home process.

See Understanding a Z2 Home for more information on how a z2 home works.

Properties of a Worker Processes Component:

name	values
com.zfabrik.component.type	**com.zfabrik.worker**
worker.process.vmOptions	General virtual machine parameters for the worker process. See the JVM documentation for details. See below for consideration on command line syntax.
worker.process.vmOptions.<os name>	Override of the general VM options above for a specific operating system. Use the OS name returned by the RuntimeMXBean to replace <os name> with. See below for consideration on command line syntax.
worker.states	Comma-separated list of target state (components) of the worker process (see also **ISystemState**). The worker process, when starting, will try to attain these target states and will try so again during each verification and synchronization.
worker.concurrency	Size of application thread pool (see **ApplicationThreadPool**). In general this thread pool is used for application type work (e.g. for web requests or parallel execution within the application). This property helps achieving a simple but effective concurrent load control
worker.process.timeouts.start	Timeout in milliseconds. This time out determines when the worker process implementation will forcibly kill the worker process if it has not reported startup completion until then.
worker.process.timeouts.termination	Timeout in milliseconds. This time out determines when the worker process implementation will forcibly kill the worker process if it has not terminated after this timeout has passed since it was asked to terminate.
worker.process.timeouts.communication	Timeout in milliseconds. This time out is the default timeout that determines the time passed after which the worker process implementation will forcibly kill a worker process if a message request has not returned.
worker.debug	Debugging for the worker process will be configured if this property is set to true and the home process has debugging enabled. Otherwise the worker process will

	not be configured for debugging.
worker.debug.port	The debug port to use for this worker process, if it is configured for debugging.
worker.inheritedSystemProperties	Some system properties set on the z2 home process via the command line (or launch.properties) are useful to be propagated to worker processes. This property can be used to specify a comma-separated list of such properties. See below for more details.

7.19.1. System Property Propagation from Home to Worker Process

Some system properties set on the z2 home process via the command line (or launch.properties) are useful to be propagated to worker processes. The property **worker.inheritedSystemProperties** can be used on worker process declaration to specify a comma-separated list of such properties. See below for more details.

Note that all properties set in runtime.properties (see #offlineMode) will be set on all z2 processes regardless. If however properties may be set depending on some starting command line, it can be useful to have them propagated as is.

By default, the following system properties will be propagated via the command line from the home process to the worker process by default:

Property	**Meaning**
com.sun.management.config.file	JMX config file. Preferrably preserve.
com.zfabrik.config	Name of config file. Defaults to runtime.properties. See #systemProps.
java.util.logging.config.file	Default Java logging configuration.
com.zfabrik.dev.local.workspace	Setting of the development repository defining where to check for “armed” modules. See also #systemProps and in particular #devRepo.
com.zfabrik.dev.local.repo	(outdated)
com.zfabrik.repo.mode	General repository access mode. If set to “**relaxed**”, repositories will attempt to work on previously cached resources in case of technical repository access failures during synchronization. If set to “**strict**” (which is the default), technical failures will lead to a failure in synchronization, even if local resources are available. This feature is useful for offline or bad connectivity situations.
java.library.path	See Java documentation.
com.zfabrik.mode	For development mode. See #systemProps

com.zfabrik.proxy.auth **https.proxyPort** **https.proxyHost** **http.proxyPort** **http.proxyHost** **proxyPort** **proxyHost** **proxyUser** **proxyPassword**	Proxy configuration. See https://redmine.z2-environment.net/projects/z2-environment/wiki/How_to_proxy_settings.

That is, the default value of **worker.inheritedSystemProperties** is:

```
worker.inheritedSystemProperties=com.sun.management.config.file,\
  com.zfabrik.config,\
  java.util.logging.config.file,\
  com.zfabrik.dev.local.workspace,\
  com.zfabrik.dev.local.repo,\
  com.zfabrik.repo.mode,\
  com.zfabrik.proxy.auth,\
  java.library.path,\
  com.zfabrik.mode,\
  https.proxyPort,\
  https.proxyHost,\
  http.proxyPort,\
  http.proxyHost,\
  proxyPort,\
  proxyHost,\
  proxyUser,\
  proxyPassword
```

7.19.2. Special considerations when specifying VM options using blanks and quotes

Virtual machine options specified using the component property **worker.process.vmOptions** or its OS specific variant will be parsed by Z2 and conveyed as command line arguments to the child process. In general, the specified command line will be broken up at blank characters (multiple in a sequence counting as one).

Special considerations apply when there is a need to define virtual machine options containing blanks or double quotes: Any text enclosed in double quotes will be treated as one “token” when breaking up command line options. That is, in order to supply an option that contains blanks, put it in double quotes.

For example

```
worker.process.vmOptions=-Dprop1=A -Dprop2=it works
```

will lead to system properties prop1=”A” and prop2=”it” and some probably problematic option “works”. In order to achieve the desired prop2=”it works”, use

```
worker.process.vmOptions=-Dprop1=A “-Dprop2=it works”
```

Finally, if you need to convey double quotes in VM options, use the backslash (“\”) escape to escape quotes. The latter works in quoted text only.

For example

```
worker.process.vmOptions=-Dprop1=A “-Dprop2=it \”works\””
```

will lead to prop2='it “works”' (using ' as string delimiter for clarity). Or, more extremely

```
worker.process.vmOptions=-Dprop1=A “-Dprop2=\””
```

will result in prop2='”'.