System-Centric Development for Java with the z2-Environment

Version 2.0

Last updated on:
April 4, 2012

Copyright 2011 by
ZFabrik Software KG
Walldorf, Germany

1. Introduction ... 3
1.1. It's all in the Repository – or what "System Centric" means ... 3
1.2. What is the z2 Environment ... 4
2. Installation and First Steps ... 4
2.1. Understanding a z2 Home ... 4
2.2. Getting a Snapshot using Git ... 5
2.3. Getting a Snapshot using Subversion ... 6
2.4. Folder Structure of a z2 Home ... 7
2.5. Starting and Sync'ing ... 8
2.6. 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 ... 14
3.2.3. Subversion Support ... 14
3.2.4. Git Support ... 15
3.2.5. Component Types and Component Factories ... 17
3.2.6. Java Naming and Directory Interface (JNDI) Support ... 18
3.3. Unit of Work and Transaction Management ... 18
4. Developing with the z2 Environment ... 19
4.1. Workspace Development Using the Dev Repository ... 20
4.2. The Eclipsoid Plugin ... 20
4.3. Unit Testing using z2 Unit ... 22
4.4. Retrieving Jars from Z2 ... 23
5. The z2@Spring Distribution ... 24
5.1. Using Spring Features in Z2 ... 24
5.2. Spring Modules in z2@Spring ... 24
5.2.1. org.springframework.foundation ... 24
5.2.2. org.springframework.security ... 26
5.2.3. Other Spring Modules ... 26
5.3. Spring Beans and Z2 Components ... 26
5.4. Modularization for Spring Applications ... 28
5.5. Advanced Notes ... 30
5.5.1. Dangling Imports ... 30
5.5.2. Static Members Holding Singletons ... 30
6. The z2@Hadoop Distribution ... 30
7. Component Type Reference ... 31
7.1. Core Component Properties ... 31
7.2. "Any" Components (core) ... 31
7.3. Component Factories (core) ... 32
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 ... 35

7.8. Java Components (core)....36
7.8.1. Class Loaders....36
7.8.2. Includes....37
7.9. JUL Configurations....39
7.10. Log4J Configurations....40
7.11. Spring Application Contexts (z2@Spring)....40
7.12. Spring Beans (z2@Spring)....41
7.13. Subversion Component Repositories (core)....41
7.14. System States (core)....42
7.15. Web Applications (z2@base)....43
7.16. Web Servers (z2@base)....44
7.17. Worker Processes (z2@base)....45

1. Introduction

The z2 Environment represents a new approach to solution development and application life cycle management in Java.

In Java development one normally assumes that some Java code, present in files ending with ".java" be present and compiled to ".class" files that, typically, get processed further into ".jar" archives or some other file format to be presented for execution to some runtime environment. In the simplest case this is the java command line, in the more complex cases this is an application server or another managed execution environment. This transformation process is called the *Build* process. It typically requires a combination of tools and cross-integrations to make sure that dependencies can be resolved, the result is reproducible, and all required sources were available.

While that may be fine for a Desktop application that is thoroughly tested and finally distributed as a binary to thousands of computers, it can be neck-breaking for enterprise software that gets constantly enhanced, repaired, and kept alive over a long period of time.

The z2 Environment eliminates these intermediate steps following an approach that is sometimes called "System Centric", thereby removing a lot of complexity and need for tedious processes. On top, once available, additional benefits show up in distribution of software, testing, and debugging.

It is noteworthy to point out that the ideas behind z2 are far from being new. Extremely successful Enterprise Platforms like SAP's ABAP server adhere to the same underlying principles.

1.1. It's all in the Repository – or what "System Centric" means

In a system-centric approach all elements of a system, that is the construct that is meant to be achieved and maintained over time, adhere to one describing and defining structure, all the time.

It is the opposite of a mass-product: In a mass-product scenario, a single concept gets implemented in possibly huge numbers using a non-trivial and potentially very specific production process.

In a system-centric approach, multiples can be achieved by copying the description and altering it to fit the requirements of the new system. It is accepted, if not intended, that copies may not be identical to their source.

For a software system this translates into having the complete system description – essentially anything that defines how the system behaves – be associated with the actual system in a one-to-one relationship. In other words: There is no additional information that is involved in turning the system description (source code, configuration,...) into a system. The source is the system.

Practically speaking, the system description is stored in files in some well-defined folder structure –

typically in a version control system repository (VCS) as we will see later. In Z2 we call such a store a *Component Repository*.

There are some implications of this approach:

As we want to make sure the system adheres to the contents of the component repository all the time, changes in the repository should be effective as soon as desired. On the other hand we do not want other external information to be part of that equation. Hence the software system should update itself at runtime. In Z2 this includes the ability to compile Java code whenever necessary.

Consequently, the typical deployment processes found in Java application servers, where a previously assembled archive is transferred to the application server infrastructure for execution, have no place in the system centric approach.

1.2. What is the z2 Environment

The z2 Environment is an implementation of what we call a system-centric approach for solution development and operation in Java.

Practically speaking the z2-Environment is a runtime environment that knows how to update itself by considering configurations stored in repositories of various technologies.

Z2 defines an extensible component model that, based on few basic paradigms and interfaces, allows to construct the full-blown application environment that is presented in this document.

The z2-Environment can be used as a multi-process server runtime as well as an embedded component runtime that is controlled from within another process. The fundamental qualities of Z2 are preserved in both scenarios.

2. Installation and First Steps

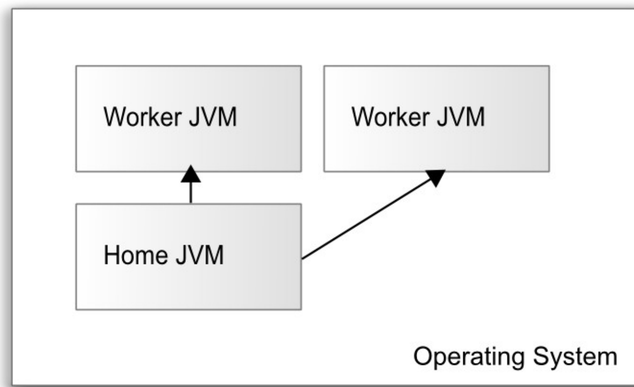
This section will give you some first hands-on experience with Z2. We will show you how to create a local installation based on the Gitorious hosted z2_base system. As a prerequisite you need to have the Git versioning system installed.

We will also describe how to get started using the ZFabrik hosted Subversion repositories, that provide a copy of the z2_base system.

But first some theory:

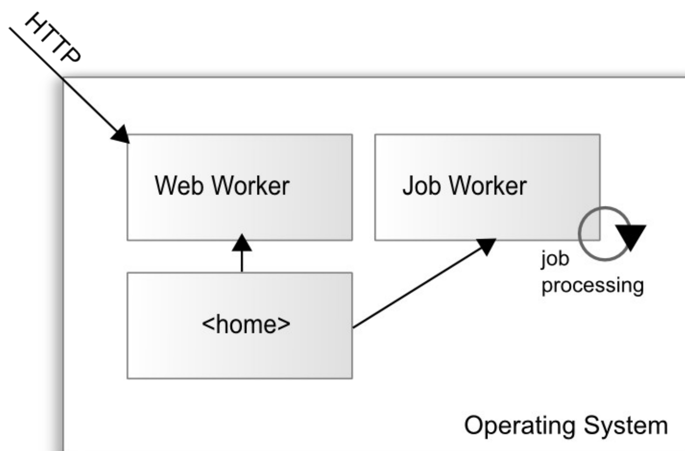
2.1. Understanding a z2 Home

As indicated above, there are different ways of using Z2. In this section we will use "Home" approach. A Z2 home is a local installation of the z2-Environment. At runtime, a home always runs a so-called "Home JVM Process" and typically one or more "Worker JVM Processes":



The separation of home and worker processes has several important implications: First of all, the home process can be kept light-weight and robust as typically no application code is executed there. This turns the home process into a reliable life cycle manager for anything else running, in particular the worker processes that can be killed and restarted if needed – e. g. because their configuration was changed. Furthermore, a home process may be configured for different configurations of worker processes thereby supporting very different topologies and load delegations within one overall configuration.

For example, in a given home setup, one process could be dedicated to serving web requests while another one takes care of processing asynchronous, possibly resource hungry processes (e.g. updating of analytical data).



Generally you install a z2 home by checking out or pulling (in Git) the main repositories (typically one for Subversion or multiple for Git).

2.2. Getting a Snapshot using Git

Using Git, getting a snapshot of the z2_base system is as easy as cloning the associated core repository. An advantage of Git in this case is that by cloning you already have a complete repository in your hands that can serve as a starting point for further spin-offs. And yet this repository is linked to its origin so that updates can be easily pulled and integrated at your convenience.

Note that z2_base consists of two repositories (core and base): One repository contains the core runtime of the z2_base system, while the other contains all other modules.

The fact that we use two repositories to model that system is due to Git's lack of partial checkouts if you will and some more. It becomes clearer when understanding that the z2_spring system,. just like other repositories, is a true super set of the z2_base system, and we do not want to lose the ability to merge changes of the base part of z2_spring (that has the repositories core, base, and spring) back into z2_base.

In our case, assuming you have Git installed and created some directory, say z2_base, this means to run

```
git clone https://git.gitorious.org/z2_base/core.git
```

in that new folder. After that you have the binary core distribution of the z2_base system. As you need more and for simplicity, there are **gitsetup** scripts in **core** that automate the act of cloning all required repositories. Change into the new **core** folder and run

```
./gitsetup.sh
```

if you are on Linux or Mac OS, or run

```
gitsetup.bat
```

if you are on windows. Once that has completed you should have folders: core, base, samples, and workspace. We already learned about core and base. Here is about the others:

samples: This folder contains some simple samples that may help you understand the structure of z2 modules.

workspace: When using Git, all your development happens in the Git repository workspace folders. In our case these are base and samples. When experimenting however you do not necessarily know already what and if you are going to add projects to the repositories and if so which ones. On the other hand (see later in #develop), if z2 can see your projects, you can round trip with ease. The workspace folder is exactly that: A workspace that will be introspected by the local z2 environment (started from core) and should be used for projects you are not sure about yet.

Note that the HEAD ref of the git repository is set to a released, but typically still maintained branch (v2.0 at the time of this writing). The gitsetup script preserves the branch choice for its checkouts.

Also note that the core repository holds a z2 core that is configured to use the very same branch name. This has some implications:

- Unless you change the system repository configuration (see #componentsAndComponentRepos) the core will use the branch of the same name from the other repositories (base and sample in this case), regardless of the checkout in their workspaces.
- See #develop on how you can nevertheless make modules from the local workspaces visible.

Another final note: The setup described in this section is the most suitable for development with Git. For production environment you would not have the local repository clones. Instead the system repository configuration would directly point to a remote repository and clone from there.

2.3. Getting a Snapshot using Subversion

When using the Subversion versioning system in conjunction with z2 you only need to check out the binary core of the system. All other resources will be fetched directly by the z2 environment.

Make sure you have the Subversion client software installed. You can use the command line or a tool like TortoiseSVN on windows. We will refer to the command line here.

To get started, create a folder and run

```
svn co http://z2-environment.net/svn/z2_base/branches/2.0/core
```

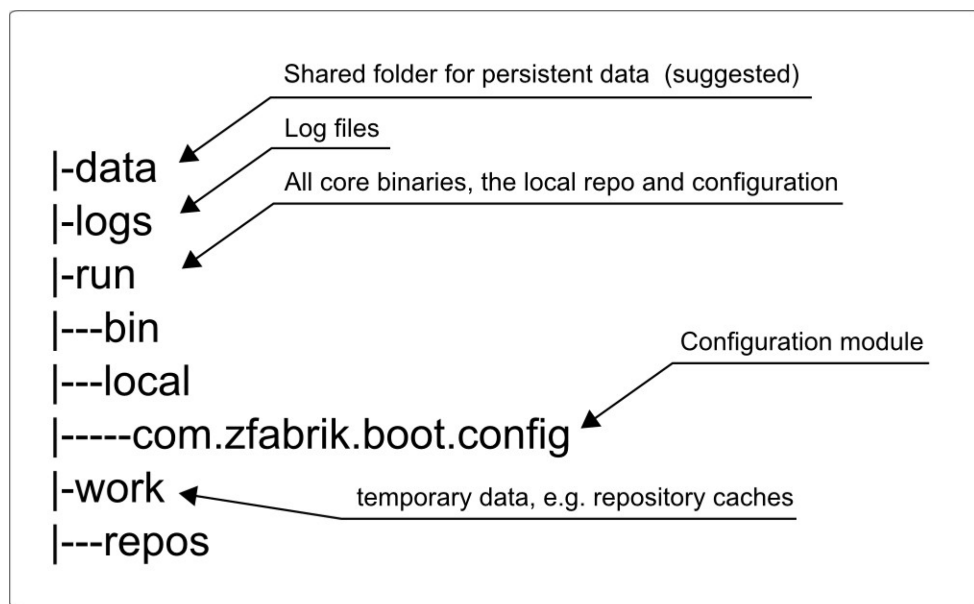
That's all you need to continue from the ZFabrik hosted SVN repositories. These are read-only. In contrast to working with Git you do not have a complete development setup at your hands now.

If you want to host your own system using Subversion you need to reproduce the structure from the ZFabrik hosted repositories. One way to achieve that is by:

1. Export core and base from http://z2-environment.net/svn/z2_base/branches/2.0/
2. Import these into your new repository.
3. Fix core/run/local/com.zfabrik.boot.config/systemRepository.properties to point to your base folder and using the correct credentials

2.4. Folder Structure of a z2 Home

Typically a home installation folder structure looks like this:



The top folders have particular meanings:

logs:

The logs folder holds (by convention) all log files generated during runtime.

data:

Applications and system extensions store data they want to preserve as file system data on the home installation. While this should be the exception, there is good use cases for that. By convention, software components should create namespaced or otherwise sufficiently unique subfolders.

The only really important convention around the data folder is that it should not be deleted without thinking about the potential data loss.

work:

The work folder holds temporary data that may be deleted when the home process is stopped. For example, the component repositories maintain locally cached resources here.

Stopping the home process, removing the work folder, and starting the home process again should not change the system behavior nor application behavior except for some extended start up time due to cache misses.

run:

The run folder contains all the binaries to start and bootstrap the home process. These binaries are not compiled on demand but must be compiled in an ordinary build process. However, in general there is no customizing or extensibility required on this level. The only exception to this rule is the configuration of the initial repositories (see #componentsAndComponentRepos).

The reason to exist for these binaries and configurations is for bootstrapping (for example: The Java compiler has to be executable before anything else can be compiled).

The subfolder bin of the run folder contains various start scripts and configuration files:

runtime.properties

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

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=\
      -Xms16M -Xmx16M -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
```

2.5. Starting and Sync'ing

In order to start z2 in server mode, change into the folder **Z2_HOME/run/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.**

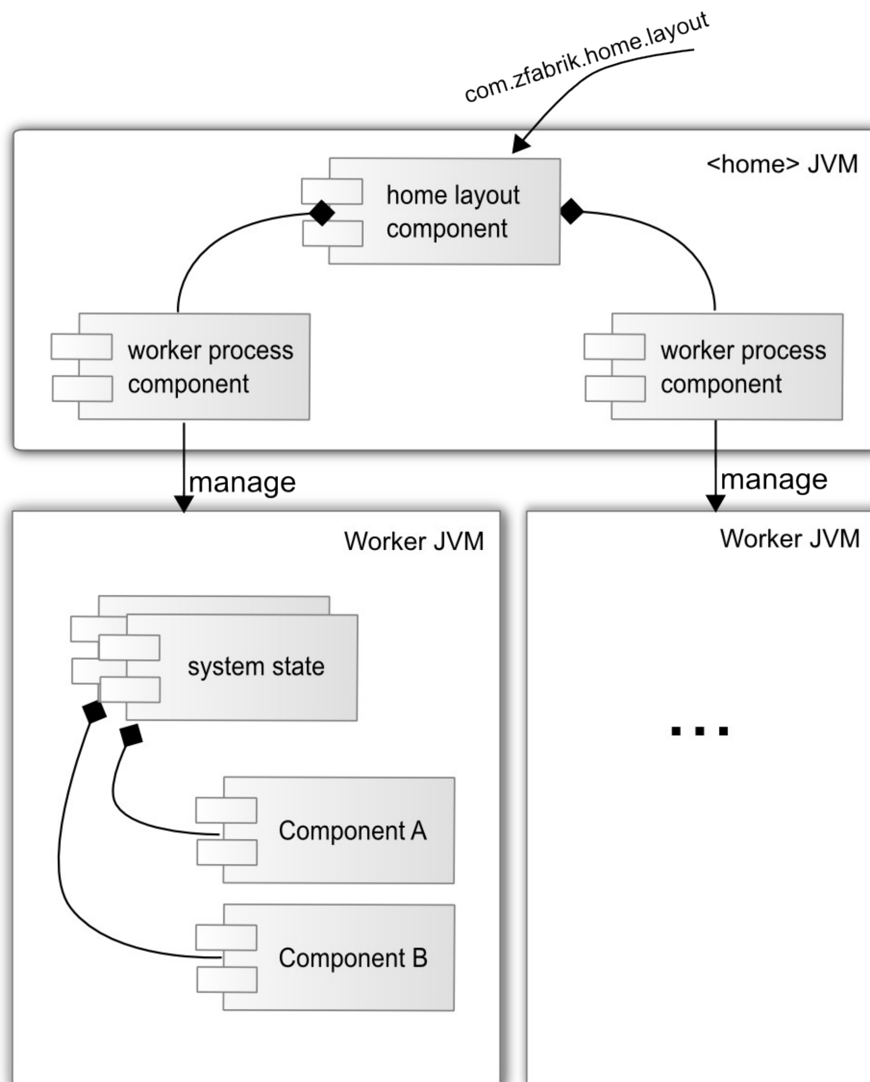
2.6. Worker Processes

As indicated in section #UnderstandingZ2Home 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 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/run/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.

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.

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

… <folder>/<local>.properties ...	Defines component **<folder>/<local>** of type of value of the property **com.zfabrik.component.type** as set in the property file **<local>.properties.**
... <folder>/<sub>/ z.properties <file/folder> <file/folder> … ...	Defines component **<folder>/<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>)**

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.

One particular repository you should be aware of the "system repository" as we call it.

When the z2 environment starts up it has a hard coded knowledge of the "local" repository. It is stored in **Z2_HOME/run/local** (see also above).

However, how does it get to any non-local content in the first place? That is what the system repository is for. You do need to touch its definition if you want to create a new home distribution that uses another "main" subversion repository for example.

The system repository is defined in

Z2_HOME/run/local/com.zfabrik.boot.config/systemRepository.properties

Obviously this is an important customization point, if you are constructing your own core distribution for your new system as this is defining where directly or indirectly (e.g. via further repository declarations) the vast majority of code and configuration comes from.

You have a set of options here. Specifically have a look at the component repository types for Subversion (com.zfabrik.svncr), GIT (com.zfabrik.gitcr), and for the file system stored repository (com.zfabrik.fscr) in section #componentTypeReference.

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 #RefFSCR 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 RefSVNCR..

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 #componentsAndComponentRepos. For example, the system repository of the z2_base distribution has the URL: http://z2-environment.net/svn/z2_base/2.0/base

3.2.4. Git Support

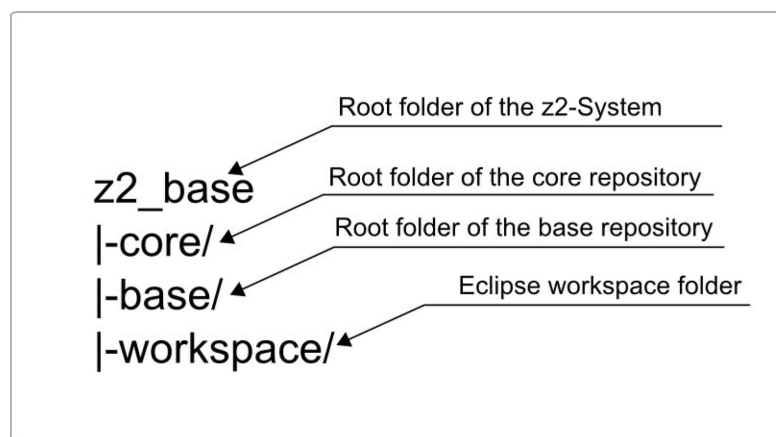
The GIT version control system (VCS) is an implementation of a distributed version control system (DVCS). As opposed to centralized VCS, such as Subversion below, in a DVCS users hold a copy (called a *clone*) of the repository content on their local environment, typically the local disk, and can execute 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.

The somewhat heated discussion whether the one approach is better than the other is beyond our scope of discussion to a large extend depending on the way you are working. We do not have a clear recommendation for you.

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 and you do not need to resort to other tools like z2's transport tool for Subversion. 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.

The z2 development setup with git

After cloning the z2_base system to your own PC and running the *gitsetup* script you will get the following file structure:



The *z2_base* directory is root directory of the whole system. In contains at least two repositories called *core* and *base* and a directory called *workspace* which is the Eclipse workspace folder.

The *core* repository contains the z2 home as described in chapter #FolderStructureOfHome, while the *base* repository contains all the basic z2 components like the web server component, the environment component and some utility components. Dedicated applications would be added to a separate repository – e.g. *webshop*.

This way all repositories can be updated individually using the git commands *git pull* and *git push*.

How does home find the components inside *base*? This is defined in the boot configuration component `core/run/local/com.zfabrik.boot.config/systemRepository.properties`. In the z2 git distribution (at github) this component looks like this:

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

# git stored component repository
com.zfabrik.component.type=com.zfabrik.gitcr

# this can also point to a remote repository like
# gitcr.uri=ssh://myserver/some/git/repo
```

```
gitcr.uri=../../../base
```

This component is of type *git component repository* as defined by the line:

```
com.zfabrik.component.type=com.zfabrik.gitcr
```

so it can deal with git repositories.

The property **gitcr.uri** defines the location of the so called *system repository*. (relative to the folder `core/run/bin` where the server is started from). In the same way you can add another git component repository (e.g. *webshop*). This is usually not defined in the *core* but inside the environment component within the *base* repository (i.e. *base/environment*).
The syntax of the git component repository URI is like this:

```
gitcr.repoFolder=<some git URIish>
```

Optionally you can also define a branch (the default is *master*):

```
gitcr.branch=<some branch name>
```

Be aware that component repositories are special in that they change the visibility of components within z2. They need to be announced to the core runtime early and in all processes, worker and home. That's why it is important to make sure your repository participates in the system state **com.zfabrik.boot.main/sysrepo_up**.This is achieved by the first line in the repository declaration:

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

Note that within the z2-svn setup the *development component repository* (see #develop) is just the system root folder (which is also used as the Eclipse workspace folder). Within the z2-git setup the workspace folder as well as the component repositories like *base* and *webshop* are one level deeper within the file hierarchy. Therefore the z2-home process must be given two dev-repositories – see also **core/run/bin/launch.properties:**

```
home.vmopts=\
      -Xms32M -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=../../../base,../../../workspace \
      -Dcom.zfabrik.dev.local.repo=../../work/repos/dev \
      -Dcom.zfabrik.mode=development
```

This fits nicely the way how the git-Eclipse plugin *EGit* works: Projects which are under version

control are only linked into the Eclipse workspace. Their physical location remains the git-working directory. EGit strongly recommends to not share the same directory for the Eclipse workspace and the git working directory. Thus one can work on projects which are either physically located inside the git working directory or which are located inside the Eclipse workspace (like throw-away test projects or new projects which aren't yet added to git). The dev component repository implementation scans both directories for changes giving the higher priority to the first one (here **../../../base**).

Working with Eclipse

When you start Eclipse the first time for a freshly cloned z2 system you should switch your Eclipse workspace to the workspace folder inside the z2 directory (e.g. `z2_base/workspace`). Now you can add the git repositories (*core* and *base*) to the “Git Repositories” view of the EGit plugin (we assume that you are already familiar with EGit).

Now import the project called “core” from the *core* repository using “Import Projects... → Import existing Projects → Next” from the EGit context menu.

Switch to the Java perspective and investigate the new “core” project. The important paths are **core/run/bin** and **/core/run/local/com.zfabrik.boot.config/**.

The project also an external tool configured that launches the z2 server inside a new window: Navigate to “Run → External Tools → External Tools Configurations → (new popup window) → Program (on the left hand side) → z2_base”. Select z2_base and click the “Run” button .

Alternatively you can start th2 z2 server from the command line (assuming you are inside the z2_base folder):

```
$ core/run/bin/gui.sh &
```

or

```
> core/run/bin/gui.bat
```

3.2.5. 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 #ResourceManagementSystem) 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.

3.2.6. 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 #RefDataSources) 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.

3.3. 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
```

or, alternatively the Transaction Manager implementation can be looked up at

```
java:/TransactionManager
```

so that in Hibernate you can use the **configuration:**

```
<property name="hibernate.transaction.manager_lookup_class"
      value="org.hibernate.transaction.JBossTransactionManagerLookup" />
```

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 transations.

4. *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 component repository* – a component repository implementation that allows you to selectively and quickly test modifications

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

4.1. Workspace Development Using the Dev Repository

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

It expects to find a component repository structure as detailed in #componentsAndComponentRepos.

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 subfolders of the folder that contains the core installation.

When using Git, or when using subversion and checking out the z2 core into your Eclipse workspace this means that the development repository scans the projects in your eclipse workspace.

Now, this is how it all ties together: When you check out a z2 project (a.k.a. module) from Subversion or from your Git workspace into your eclipse workspace you can simply put an empty file called **LOCAL** into the project's root folder and the project and all its components will be picked up 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 components and via the system property **com.zfabrik.dev.local.modulesLevel** you can specify at what folder depth it expects to find modules (that hold a **LOCAL** file and component definitions according to the component repository file system structure mentioned before).

4.2. The Eclipsoid Plugin

The Eclipsoid plugin 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 plugin provides a number of useful utilities for working with z2. The most important functions are however:

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)

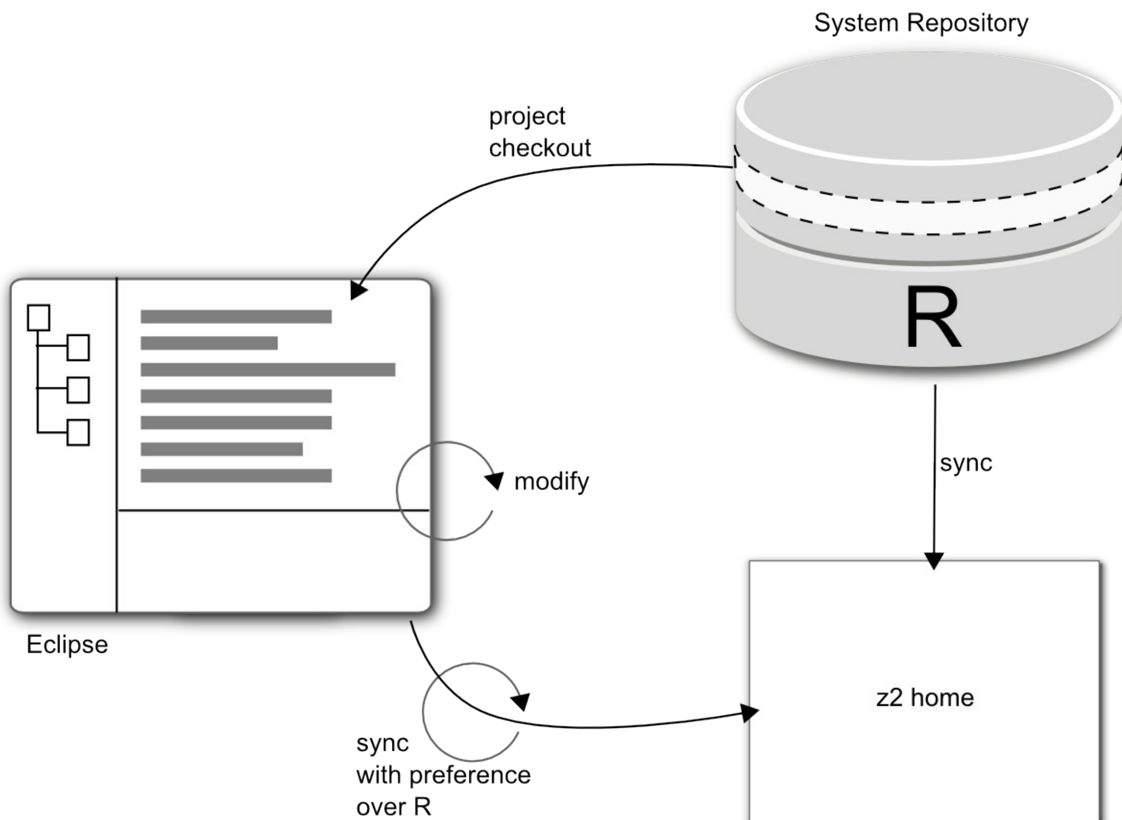
The Eclipsoid fixes an important problem in development of larger software systems in IDEs like Eclipse:

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 solves. Upon Resolve, any Eclipse Java project that has the Eclipsoid

classpath container in its .classpath will be introspected for Java components (see #RefJavaComponents), and if found, their references will be resolved on the server-side and required Java definitions will be downloaded and provided to the project.

That means: In order to work on a single project, from a much larger 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.



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

Furthermore the Eclipsoid can “arm” and “disarm” projects. Arming a project simply means to put an empty **LOCAL** file in it, and disarming means to remove it. See the previous section for more details on the Dev Repo. Projects that are armed have a green Halo around the Z decoration in the project view.

4.3. 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. This makes a huge difference. Standard JUnit tests run in an environment that often has little to do with the tested codes “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 firstly abstracted out from the tested code and secondly “mocked”, that is simulated one way or the other, into the test assembly of things.

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).

See URL for more details on the two annotations you need to use when using z2Unit.

To see how this works in Eclipse

1. create a Z2 Java project **my_tests** in your Eclipse workspace. Make sure it is arned.
2. add a test reference to **com.zfabrik.dev.z2unit** to your Java component **my_tests/java.** The z.properties should look like this:
3. create a test class in src.test of your Java component like the following:

```
package tests;

import org.junit.Test;
import org.junit.runner.RunWith;
import com.zfabrik.z2unit.Z2UnitTestRunner;
import com.zfabrik.z2unit.annotations.Z2UnitTest;

@RunWith(Z2UnitTestRunner.class)
@Z2UnitTest(componentName="my_tests/java")
public class AZ2UnitTest {

	@Test
	public void someTestMethod() {
		System.out.println("This is just a test method");
	}
}
```

4. Resolve using the Eclipsoid plugin (see above)
5. Right-click and choose Run-As / JUnit Test

There is one important pre-requisite when running a z2Unit Unit Test class. You need to have all dependencies of the test class resolved so that JUnit can resolve the test class locally (although all the action will happen elsewhere) - which is accomplished using the plugin.

So, if you want to automate 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.

4.4. 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 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="${z2home}/run/bin/z_embedded.jar"
      classname="com.zfabrik.launch.MainRunner" fork="true">
      <!-- general config →
      <sysproperty key="java.util.logging.config.file"
             value="logging.properties" />
      <sysproperty key="com.zfabrik.home" value="${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

5. *The z2@Spring Distribution*

The z2@spring distribution is a super set of the z2@base distribution. It contains the popular Spring Framework (currently in version 3.0.5), some integration features with the Spring Framework, notably the AspectJ support, modularization features such as component sharing between application contexts.

You can get it via Git or Subversion access just as it can be done for z2@base and as it has been described in #snapShotGit and #snapShotSVN respectively.

In this case the relevant URLs are

https://git.gitorious.org/z2_base/core.git

and

http://z2-environment.net/svn/z2_spring/branches/2.0/core

5.1. Using Spring Features in Z2

This section highlights the integration of Spring with z2 based on the integration features implemented in the project "com.zfabik.springframework" as well as by the module separation choice for the Spring framework.

All example snippets below are taken from the (very simple) example application consisting of the projects

samples.spring.simplemodules.frontend

and

samples.spring.simplemodules.services

that you can find in the z2@Spring repository.

This example illustrates the use of integration features with the z2 environment such as:

- The Component Factory Bean.
- Exposing Spring beans and Spring application contexts as Z2 components.
- Use of the Spring AspectJ aspect with Z2.

5.2. Spring Modules in z2@Spring

For use within the z2-environment and reflecting its own modularization, the Spring framework has been split into several z2 modules. These modules essentially consist of one or more of the original Spring modules. Those offer choices of Spring capabilities and extensions that you can make according to your use cases – and so do the corresponding z2 modules.

All these modules have names starting with **org.springframework.**

5.2.1. org.springframework.foundation

This module contains the most commonly used Spring libraries and exposes them by its Java component.

In addition it provides Java *files* components for inclusion into the referencing component.

You do not need to understand the details of what that means just now and can simply continue reading. Eventually however it will pay off to understand theses details. Please visit #RefJavaComponents for details on including files components from a Java component. Also visit #SpringAdvancedNotes below for more details on the why and what of classloading issues when using Spring in Z2 (and actually in any other non-trivial class loading setup).

The files component **org.springframework.foundation/context.support** provides a number of adapter classes that allow simple use of various widely used frameworks such as the Velocity template engine in Spring applications.

This z2 component must be included (and cannot simply be referenced) as the adapter classes in Spring Context Support require to *see* the relevant third-pary libraries at runtime – depending on the

use case. In short, the library must run in the using class loading context to resolve the type dependencies. Otherwise the Spring foundation project would need to have all the missing dependencies (and hence z2@Spring would have to hold ALL supported third-party libs.

Similarly the component **org.springframework.foundation/aspects** provides the Spring AspectJ aspect as a *"files"* component.

In this case, it is the way the aspect refers to the Spring application context that mandates to use it as a "classpath singleton" from the using component. Or simply put: It cannot be shared on the classpath.

In order use Spring at you must reference **org.springframework.foundation** (or **org.springframework.foundation/java**). In your Java component this would require you to add the reference to the private or public references. For example like this:

```
java.privateReferences=\
      org.springframework.foundation
```

If you want to use the Context Support libraries, you must additionally add the corresponding include declaration, for example

```
java.privateIncludes=\
       org.springframework.foundation/context.support
```

Note: If your project exposes an API, it is recommended to reference the context support component only as a private reference, that is, from the Java component's implementation part as above.

The Spring AspectJ aspect, altogether a very convenient but not exactly trivial matter, allows you to use annotation-based Spring configuration even for Objects that are not instantiated from within the Spring application context, such as Servlets, helper classes etc.

To enable it you must

1. Include **org.springframework.foundation/aspects**. As above it is strongly recommended to do so only from the implementation part of a Java component, i.e. via a private include (see also #SpringAdvancedNotes)
2. Additionally reference **javax.persistence**. The Spring aspect has a built-in dependency on JPA API classes.
3. Make sure the AspectJ compiler is invoked by adding the line

```
java.compile.order=java,spring_aspectj
```

to the Java component properties (see also ICompiler).

In the example application that uses the Spring aspect, the z.properties file of the frontend module looks like this:

```
com.zfabrik.component.type=com.zfabrik.java
```

```
java.privateReferences=\
      com.zfabrik.servletjsp,\
      org.springframework.transaction,\
      org.springframework.web,\
      com.zfabrik.springframework,\
      samples.spring.simplemodules.services

java.privateIncludes=\
      org.springframework.foundation/aspects
```

Annotation based configuration is used in the controller servlet implementation of the web app component in **samples.spring.simplemodules.frontend/web**. While the servlet is instantiated by the web container, that is outside of Spring's direct control, the Spring aspect still associates it with the application context of the web app as we the class is annotated correspondingly:

```
@Configurable
public class ControllerServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    @Autowired
    private IComputationService computations;

…
```

5.2.2. *org.springframework.security*

The Spring Security project consists of some exposed Java libraries for shared use and a lot of “files” components. The files component serve similar integration or AspectJ purposes as described above for the Spring foundation project. In other words, if you want to use the Spring Security LDAP support, you should include the corresponding files component from the Spring Security project.

5.2.3. *Other Spring Modules*

Other modules starting with org.springframework projects expose the corresponding Spring module as shared Java components including required references.

5.3. Spring Beans and Z2 Components

When using the Spring framework, application components are abstracted as *Spring Beans* that belong to an *Application Context*. Spring Beans can be configured, in particular with respect to their use of other beans, using the application context XML declarations or using annotations.

The Spring application context lives within a Web application or, conveniently in z2, within the context of a Java module. Spring beans are hence abstractions of “application components” as they get used within a Web application or within a Java module and hence shared among all components of a module. Z2 components are typically more coarse grained and usable across modules. It is this separation line where z2 and the Spring framework meet.

Components in z2 can be looked up using the IComponentsLookup interface or JNDI. Alternatively

you can map Z2 components directly into your Spring application context using the Component Factory Bean, **com.zfabrik.springframework.ComponentFactoryBean,** that is exposed by the **com.zfabrik.springframework** module.

Given a z2 component **samples.spring.simplemodules.services/computations** that implements the interface **samples.services.IComputationService** you can map it into your application context for the given type by adding the XML snippet to your application context:

```
<bean id="computations"
class="com.zfabrik.springframework.ComponentFactoryBean">
      <property name="componentName"
            value="samples.spring.simplemodules.services/computations"/>
      <property name="className"
            value="samples.services.IComputationService"/>
</bean>
```

Conversely you may want to make a Spring bean from your application context available as a z2 component, so that it can be looked up from other Z2 components (such as another module's application context even) or so that it may participate in system states life cycle for example.

In that case you need to first make the application context known to z2. That is technically necessary so that beans from it can be exposed while the application context is still a singleton.

When looking up a component that is implemented by a Spring bean, its application context will be initialized first – if not already.

The following component declaration declares an application context to Z2:

```
com.zfabrik.component.type=org.springframework.context
#
# context config location is where the context is
# actually defined.
#
context.contextConfigLocation=classpath:META-INF/applicationContext.xml
```

Note: Declaring a system state participation binds the application context to the system state life cycle and hence you can easily run application contexts independently from web apps. (One prominent use case of this is to execute scheduled jobs on different worker processes than those serving a web frontend.)

To do so, as above, it is necessary to provide the application context XML file as a class path resource of a Java module (typically on the implementation side, i.e. in **src.impl**).

Having an application context bound as a we can expose it's beans as z2 Component:

```
com.zfabrik.component.type=org.springframework.bean
#
# the context that defines the bean (more than one
# bean can be exposed like this)
#
bean.context=samples.spring.simplemodules.services/applicationContext
#
```

```
# the bean name
#
bean.name=computations
```

5.4. Modularization for Spring Applications

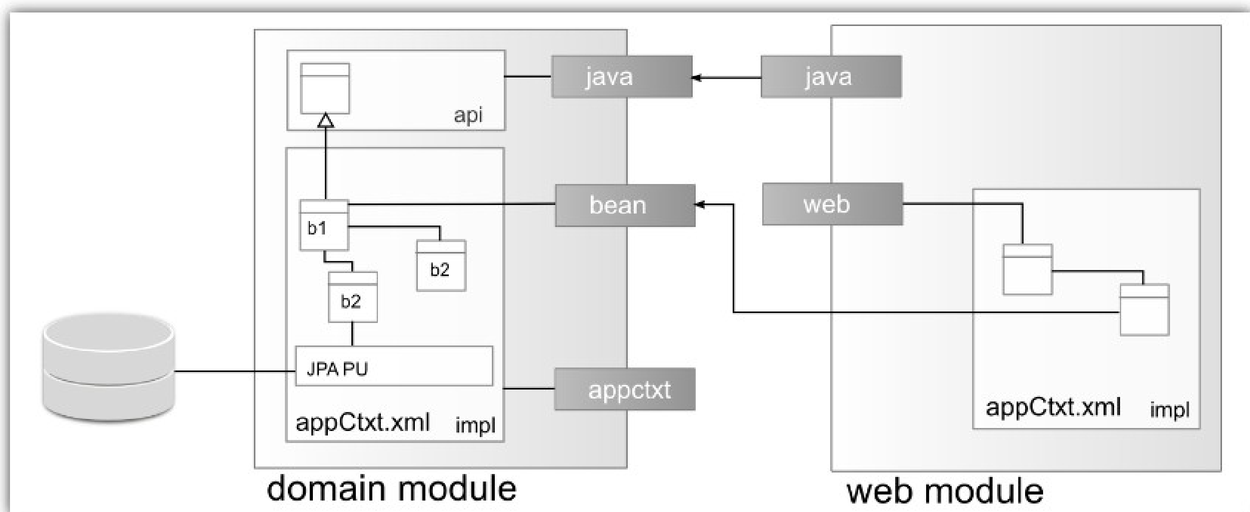
Using the integration features introduced in the previous section, the z2 component model can be conveniently used to built re-usable, shared modules for the Spring-based applications.

Compared to assembly of monolithic applications from build artifacts this approach has several benefits: For example a module that uses Java Persistence API (JPA) persistence and that is used by several web applications will need to initialize only once – saving significant time and memory resources. Spring beans implementing application services can be singletons on the VM level and optimize resource usage, e.g. for caches and connection pooling.

Building resuable modules is very easy. Essentially you declare an application context as a z2 component as explained above and expose beans that implement services or provide access to entity managers and so on by bean components to expose features implemented in a z2 modules.

In other words: z2 modules are the right choice as modules of a solution that provide any number of re-usable or private resources, public APIs and private implementation.

In the illustration below, the domain module is modeled as a z2 module exposing a Spring bean **bean** of the application context **appctxt** that is defined as a Spring application context with a JPA persistence unit. The web module simply contains a web app **web** and its Java component has a private reference to the domain module.



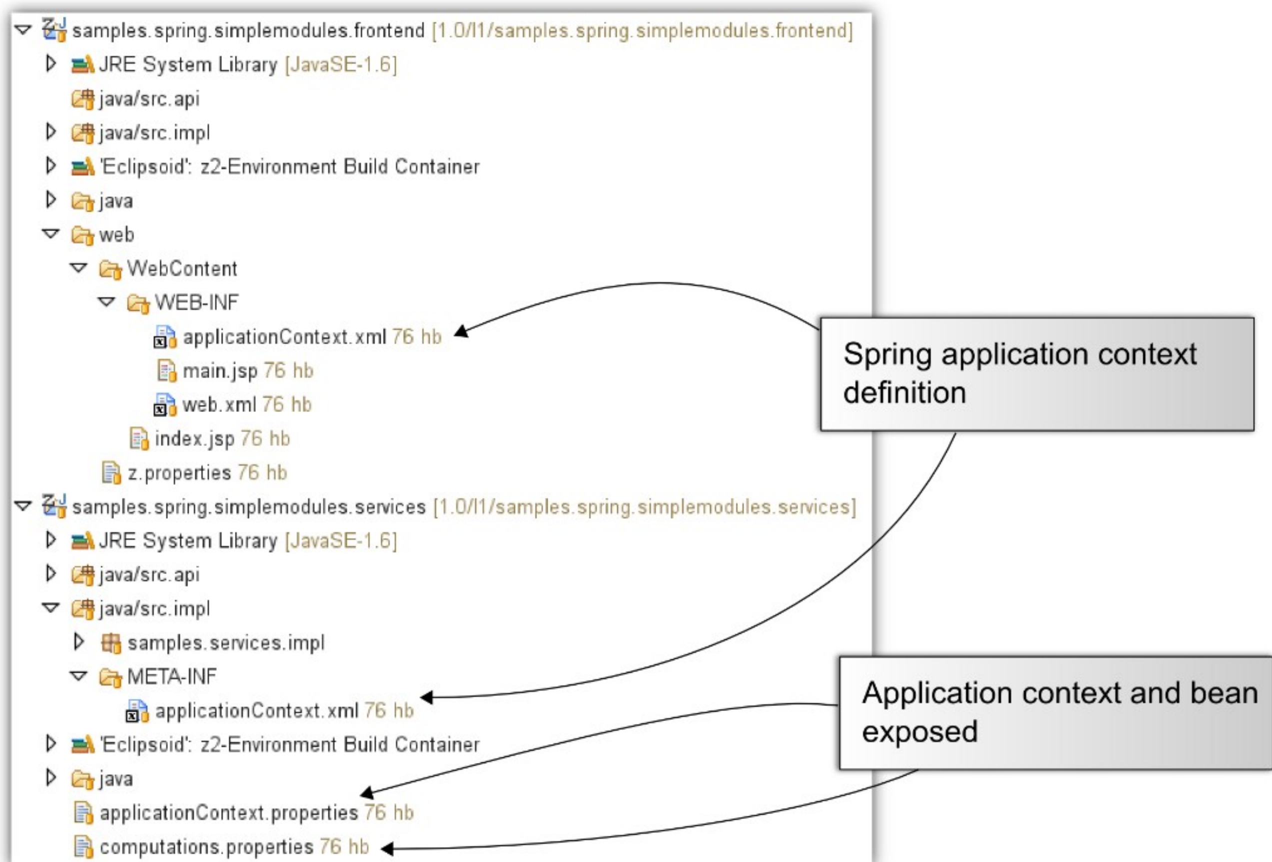
The sample application in the z2@Spring repository has two modules:

1. A service module **samples.spring.simplemodules.services** that provides reusable services and has an API. It could also contain data source definitions, schema migrations, schedulable jobs etc.
2. A frontend module **samples.spring.simplemodules.frontend** that contains a web app exposing functionality implemented over the service module.

In **samples.spring.simplemodules.services**, the **computations** component is implemented by a

spring bean from the application context of the module, defined in the implementation part of **samples.spring.simplemodules.services/java**. It implements the service interface samples.services.IComputationService.

The following illustration shows the module structure of the sample application:



The Web application **samples.spring.simplemodules.frontend/web** has an application context that binds the other modules component via the Z2 Component Factory bean:

```
<bean id="computations"
      class="com.zfabrik.springframework.ComponentFactoryBean">
      <property name="componentName"
            value="samples.spring.simplemodules.services/computations"/>
      <property name="className"
            value="samples.services.IComputationService"/>
</bean>
```

5.5. Advanced Notes

Some features of the Spring framework have slightly non-trivial prerequisites. In general these require that Spring provided libraries are used within the context of other Java components and that requires some explanation:

At runtime, Java components in Z2 have two class loader instances. One for the API definitions of the component, the other for the implementation definitions of the component (see also

#RefJavaComponents) The latter is not visible to any referencing component, while the former is. Visibility is achieved by class loader delegation. That is, the class loaders of a referencing Java component will first delegate to the API class loader of the referenced component when looking for a type before checking their own resources.

The delegation sequence is always strictly along the references.

When we use the term shared library or shared Java component, we are referring to a Java component that can be referenced by others to provide some functionality. At runtime, the types of that Java component are loaded exactly once into the VM's memory.

As a side-effect, static class members for example will be shared amongst all usages.

The alternative to sharing a Java component is to "include" a files component. When a Java component references a component of type **com.zfabrik.files** or **com.zfabrik.java**, libraries and class files of the referenced component will be copied into the referencing component (depending on the reference either into the API or the implementation part). In that case, the types provided by the files component are used in the context of there referencing component and may be loaded several times.

Considering Spring there are two cases that mandate a non-shared use of libraries.

5.5.1. Dangling Imports

Some Spring libraries provide integration of Spring with a variety of third-party libraries. When those libraries were compiled, all those third-party libraries were present on the compilation class path. Java's lazy linking paradigm allows to use those libraries without the presence of the third-party libraries as long as visibility of those types is not necessary yet.

One example of such a library is the context support Spring module. See above for more details.

5.5.2. Static Members Holding Singletons

Another reason that mandates in context use is module specific static initialization. In other words there is a class that holds static, class-level data that is specific to the using context.

One example of such is the Spring AspectJ context that points to its underlying application context by a class variable. As a consequence, sharing these classes between modules with different application context would lead to confusions about what application context is used during the application of the Spring aspect. That is why the Spring AspectJ library should always be included by a private reference, and the implementation should use only one application context as far as AspectJ configuration is concerned.

6. The z2@Hadoop Distribution

That distribution is not yet available but so hot that we believe it should be mentioned. Using the Hadoop integration features you can define Map Reduce jobs simply as z2 components, using all domain types, data source configuration, essentially anything that makes your system, without extra deployment or build steps that would be job dependent.

Contact us, if you would like to learn more about it.

7. *Component Type Reference*

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 may 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 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.

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 [#componentsAndComponentRepos] 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 constructer 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 #JNDISupport) of the z2-environment can be used make these datasource accessible typical 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 data source. 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

name	values
	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	The data source implementation class, if using NativeDataSource (see below).

7.4.1. *Data Source Specific Configuration*

When specifying a native data source 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.

A typical ZFabrikPoolingDataSource configuration looks like this:

```
#
# MySQL driver configuration
#
ds.prop.driverClass=com.mysql.jdbc.Driver
ds.prop.user=<db user name>
ds.prop.password=<db user password>
ds.prop.url=jdbc:mysql://<db host>:<db port>/<database name>?
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 #componentsAndComponentRepos. 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 #develop), 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**
fscr.folder	Store folder, i.e. the file system folder that holds the actual resources to run the repository over..
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.

See also #gitSupport.for more details on Git support.

Properties of a Git Component Repository Component:

name	values
com.zfabrik.component.type	**com.zfabrik.gitcr**
gitcr.uri	The URI to the Git repository to clone from. This can be an absolute path, a local path relative to .../run/bin, or a remote URL. See the Git documentation for examples.
gitcr.priority	The priority of the repository. Defaults to 500.
gitcr.branch	The branch to clone. Defaults to **master**.

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 #UnderstandingZ2Home 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 #FolderStructureOfHome)

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. Class Loaders

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

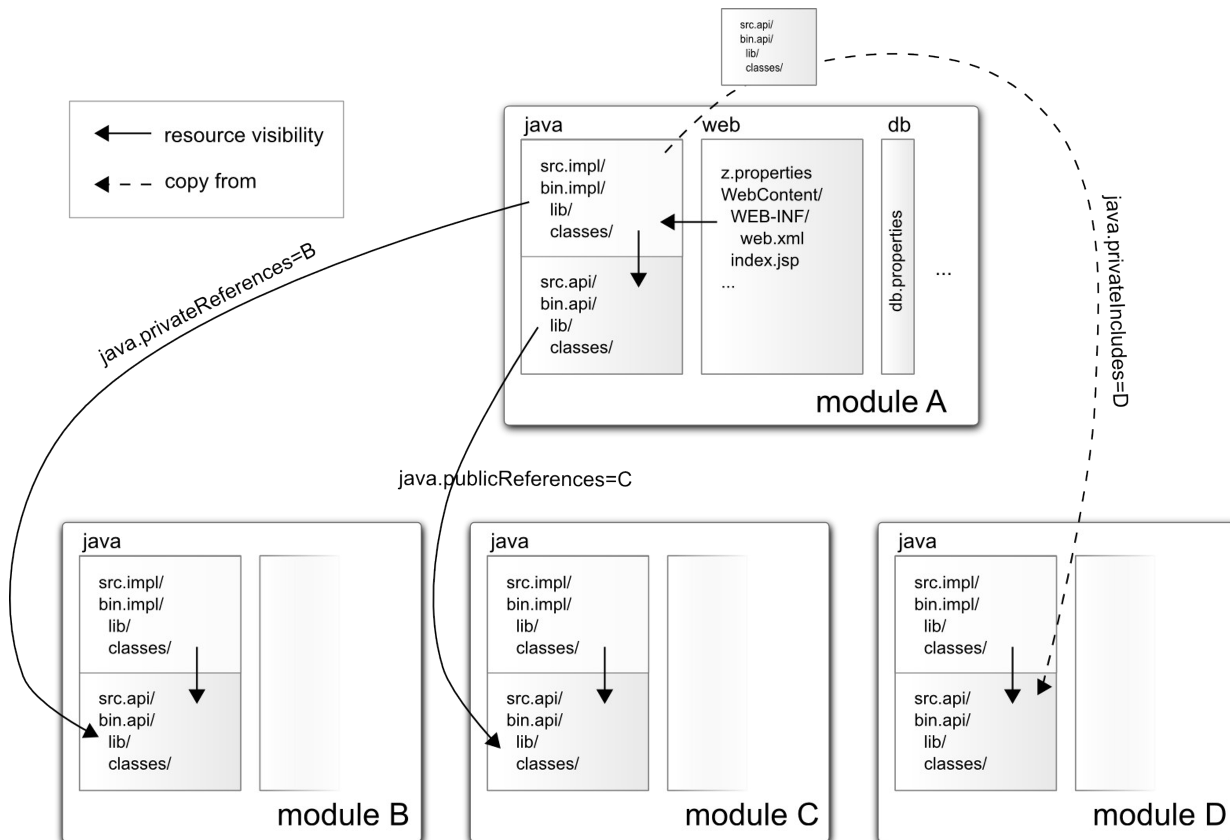
Another important mechanism supported by the Z2 Environment is so called "Java includes". The references feature described above allows sharing of types and class path resources without duplicating them at runtime.

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.privateReferences	Points to another java component whose public types will be shared with this one (and maybe others) Nothing referenced as

name	values
	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.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 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.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

The standard logging implementationcontained 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. Log4J Configurations

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**

7.11. Spring Application Contexts (z2@Spring)

As outlined in #SpringZ2Components, it can be useful to expose a Spring application context as a z2 Component, either because you want to expose beans of the context as z2 components for sharing across modules or because you want to initialize the application context based on worker process target state configuration or other z2 life cycle functions.

Properties of a Spring Application Context Component:

name	values
com.zfabrik.component.type	**org.springframework.context**
context.contextConfigLocation	Defines where to look for the context definition. If prefixed by **classpath:**, the module's Java component will be searched using a **ClassPathXmlApplicationContext** (see the Spring Framework documentation).Otherwise the location it will be supplied to **FileSystemXmlApplicationContext** (see the Spring Framework documentation)and will be search relative to the component's resource folder.

7.12. Spring Beans (z2@Spring)

The Spring bean component type exposes a Spring bean from a named application context component (see right before) as a z2 component. See also #SpringApplicationContexts.

When asked for a specific implementation via the **IResourceHandle** interface (or equivalently via the **IComponentsLookup** interface), the component's resource implementation will simply check the bean class for compatibility and either return the bean instance, in case it can be casted, or return **null**, if it cannot be casted.

Properties of a Spring Bean Component:

name	values
com.zfabrik.component.type	**org.springframework.bean**
bean.context	Name of the context component (of type **org.springframework.context** (see above) that defines the bean.
bean.name	Name of the bean in the context above

7.13. 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 #componentsAndComponentRepos 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.svncr.mode=relaxed
```

in **launch.properties**. 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.

7.14. 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 can be 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.

Using the component dependency feature (see #CoreComponentProps) you could choose one of the components of a sub-scenario as a leading component and list all other required components.

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.

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.component.dependencies=\
      mymodule1/comp1,\
      mymodule2/comp2
```

Or 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.

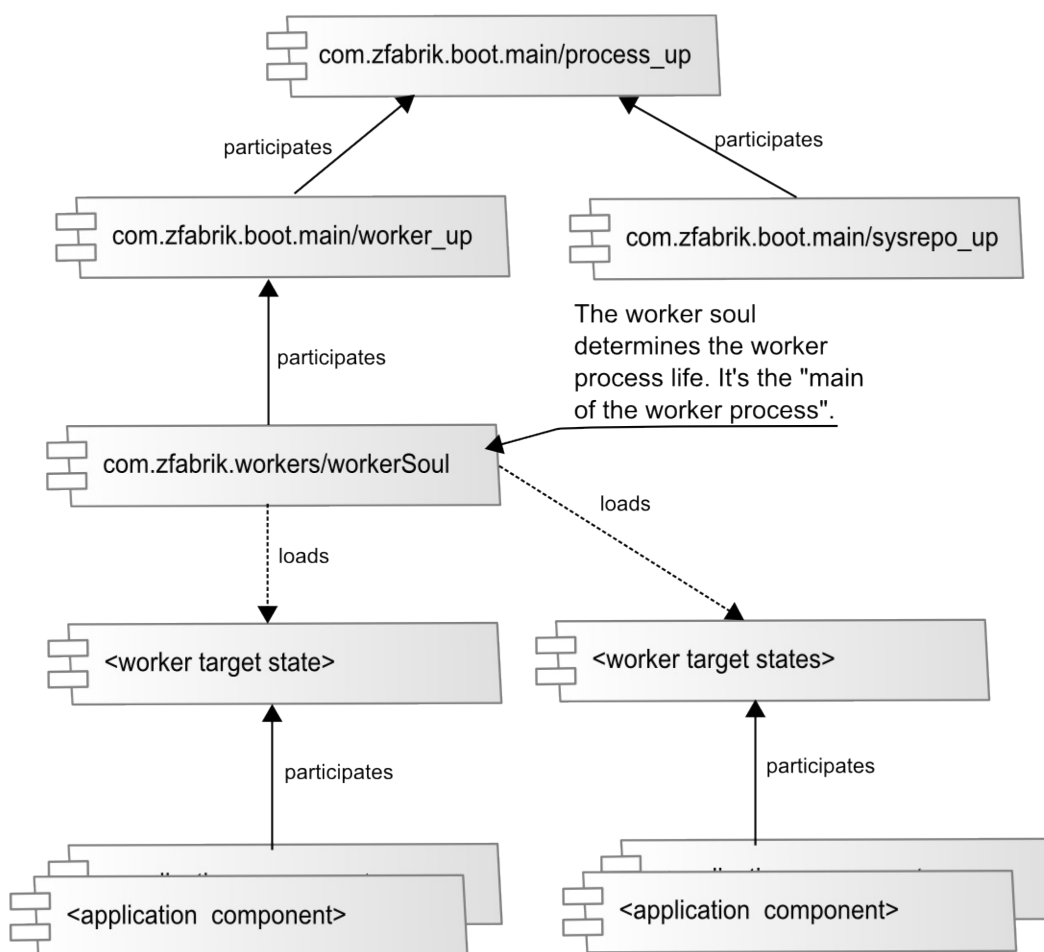
Larger scenariios may need more fine-grained scenario partitioning.

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**

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



7.15. 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.

7.16. 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

name	values
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

7.17. 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 #UnderstandingZ2Home 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.
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.
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.