Handling Application Properties

Simplify Application Customization in Different Environments

Tim Wellhausen

kontakt@tim-wellhausen.de
http://www.tim-wellhausen.de

Martin Wagner, Gerhard Müller

martin.wagner@tngtech.com
gerhard.mueller@tngtech.com
http://www.tngtech.com

January 10, 2010

Introduction

Applications are often deployed in different environments, e.g. on developers' machines, in test or production. In each of these environments, certain configuration settings differ. Examples for such settings are:

- Database logins/passwords
- Access parameters of remote systems
- Email addresses or mobile phone numbers for administrative notifications
- Localization files and configuration
- Interval settings of periodically recurring jobs
- On/off switches of specific sub-modules
- Directories for program resources

This paper aims at evolving a language of patterns dealing with these application properties. It contains patterns dealing with flexible ways of loading properties, providing common properties for multiple deployments and enforcing the explicit setting of some properties. Using the patterns facilitates maintenance of properties, allows for simplified refactoring and gives the possibility to keep varying environment settings under version control. As property storages, property files and database tables are discussed.

The paper starts with the probably well-known PROPERTY LOADER pattern and elaborates on its particular problems and shortcomings. The paper then discusses patterns that improve the handling of application properties in specific situations. At the end of the paper, a reference implementation in pseudo code shows how the patterns interact in practice.

Illustration 1 shows an overview of the patterns and how they depend upon each other. Please note that an arrow from pattern A to pattern B denotes that pattern A provides a context for pattern B.
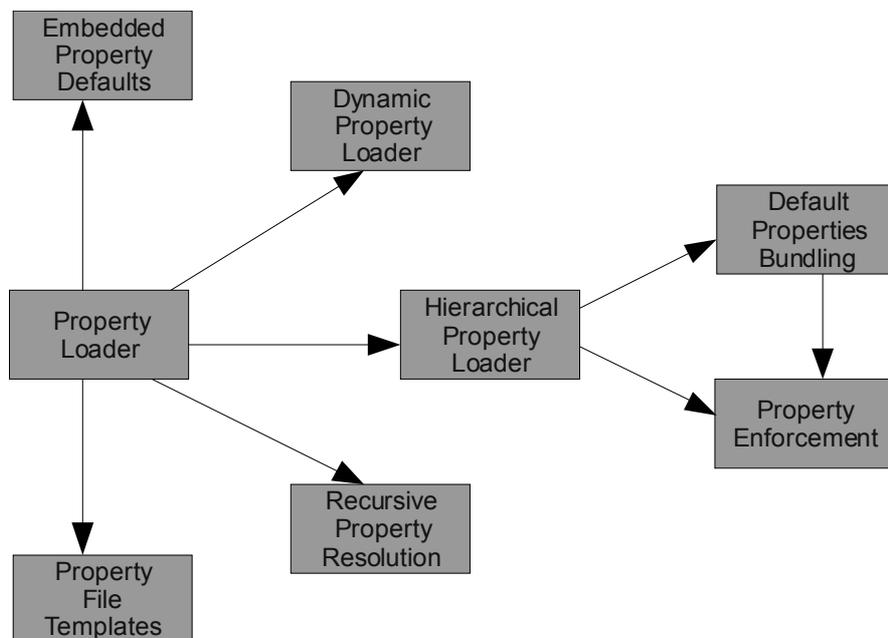
Illustration 1: Dependencies between patterns described in this paper.

Property Loader

Context

A software application may be deployed and run in different environments. An application that is used internally in a company, for example, may need to run in different environments for development, testing, integration, and production. An application that is sold as a product may be deployed at many different customer sites.

Problem

Varying environments usually mandate different behavior and configuration of the application. This may include different passwords or connection parameters, the configuration of caches, or the email addresses of people to notify in case of errors.

How can you adopt an application to varying environments?

Forces

The following *requirements* make the problem difficult:

- *Easy to use as developer.* There should be an easy-to-use programming interface to determine the value of a property with respect to the current environment at runtime. The logic on how to retrieve a property value should not be distributed all over the application.

- *Changing behavior after deployment*: It should be possible to change the behavior of a deployed application without modifying the code and redeploying the application.

Solution

Extract all configurable properties from the source code into an external storage and use a PROPERTY LOADER to load and provide access to the property values at runtime.

Analyze the application to find those parts that need adoption to different environments. Create a property, that is a key/value pair, for every such part and collect the properties in a property storage.

Create a PROPERTY LOADER that reads the properties from the storage at start-up time and makes the values of all properties available to the application at runtime via a dedicated API. The application may query the PROPERTY LOADER any time for the value of a specific property to change its behavior accordingly.

Deploy the property storage together with the application binaries and change the property values in every installation so that they fit the needs of the respective environment.

Property storages can be every form of persistent memory that is able to store pairs of property names and values. The most common property storages are text files and database tables.

A simple text file as property storage typically contains a key/value pair on each line of text as shown below:

```
db.username=user
db.password=secret
```

Accordingly, a database table for properties typically contains at least two columns: one for the property name, which might be used as primary key column, and one for the property value with both columns containing text.

The following Java interface shows an example API for a minimal PROPERTY LOADER that uses a property file as property storage.

```
public interface PropertyLoader
{
  /**
   * Loads all properties from the file with the given name.
   */
  public Map<String,String> loadProperties(String propertyFileName);

  /**
   * Returns the value of the property with the given key if
   * that property exists, null otherwise.
   */
  public String getValue(String key);
}
```

Consequences

A PROPERTY LOADER offers the following *advantages*:

- The PROPERTY LOADER provides a centralized access point for all application properties and handles the loading of the properties.

- All relevant properties of an application are located in a single property storage. It is therefore easy to check and manage all properties at a glance.

- By modifying the externalized properties of a deployed system, you can change the behavior of the system without redeploying the system. In particular, it is not necessary to change the source code of the application to adopt it to another environment.

The PROPERTY LOADER also has the following *liabilities and shortcomings*:

- Application developers often cannot foresee exactly which parts of an application need to be adaptable in different environments. If a new environment or a new requirement mandates a change in the behavior of an application for which no properties yet exist, the source code must still be modified.

- By relying exclusively on an external property file, an application may be vulnerable to incompatible manual changes in the property file resulting in runtime errors. By using EMBEDDED PROPERTY DEFAULTS, the application may protect itself against missing or corrupt property values.

- Every application update may introduce new properties that need to be set or adjusted in a local installation. Bundle PROPERTY FILE TEMPLATES with your application binaries for easier and more robust local customization of the application.

- Some applications need different property values not only for different environments like development, integration, and production but also for different hosts

and different users (e.g. an application developer needs different property values while developing the application than a user needs). Maintaining a separate property storage for each possible combination of property values can be very cumbersome. In that case, consider creating a HIERARCHICAL PROPERTY LOADER to maintain a storage with default properties and separate storages with differing properties for each environment or user.

- After an application has been successfully deployed, new properties may be introduced at any time when the development continues. When an application upgrade is to be installed, care must be taken to also deploy the new properties. This may prove exceptionally difficult if some properties of the deployed system were manually changed. To simplify this problem, consider using a HIERARCHICAL PROPERTY LOADER with DEFAULT PROPERTIES BUNDLING.

- Once a property key has been defined and is in use, it is very difficult to change the name of the key because developers don't have access to the property storages of local installations. Using a HIERARCHICAL PROPERTY LOADER with DEFAULT PROPERTIES BUNDLING also lessens this problem a little, in particular if used in combination with RECURSIVE PROPERTY RESOLUTION to provide aliases.

- In order to let changed property values come into effect, the application needs to be restarted, which is not feasible in many cases. DYNAMIC PROPERTY LOADING may help.

- Sometimes, sets of property values need to be changed at once, i.e. if one property value of a set is changed, all other properties also need to be changed consistently (e.g. host/username/password for a remote connection). Consider RECURSIVE PROPERTY RESOLUTION to manage property sets.

- If a database table is used as property storage, obviously, the database connection parameters itself must be stored in a different place. As such, a hybrid approach combining files and databases as property storage is preferable.

Known Uses

The Java platform brings its own standard properties API that can be used to easily access application properties stored in files [4]. The Log4J logging framework uses this framework to allow a flexible configuration [11].

Early versions of the Windows platform made extensive use of `.ini` files, holding properties in files. Starting with the Win32 API, this approach has been replaced by the Windows registry [5].

Most applications for the UNIX platform are configured via property files, usually stored in the `/etc` file system tree or in dot files.

Embedded Property Defaults

Context

In an application, properties are externalized from the source code by using a PROPERTY LOADER that provides an API to retrieve the values at runtime.

Problem

A property storage may not be consistent with the source code at all times: When the content of a property storage is edited, for example, errors may arise by accidentally changing the key of a property. Or a new property may be introduced to the application but cannot be immediately added to the property storages of all installations. The application code should be made robust against such errors.

How do you make sure that an application always receives a valid value when requesting a property from the PROPERTY LOADER?

Forces

The following *constraint* applies:

- *Robustness*: The application code that queries a property value should not cope explicitly with non-existing property values as this puts a burden on individual programmers.

Solution

Extend the PROPERTY LOADER and provide a method that takes a default value and returns that value when the requested property does not exist in the property storage.

By calling such a method, an application makes sure that it always gets a valid value from the PROPERTY LOADER regardless of the content of the property storage.

The following example extends the Java interface given earlier as an example in the PROPERTY LOADER pattern.

```
/**
 * Returns the value of the property with the given key if
 * that property exists, the given default value otherwise.
 */
public String getValue(String key, String default);
```

Consequences

The given solution has the following *advantages*:

- The application is more robust against inconsistencies in the property storage.
- It is not necessary to set values for all properties in every installation.

The solution also has some *disadvantages*:

- Application developers must take care to always use the API method that takes a default value.

- The default value itself is embedded in the source code, which may not be desirable. If you use a HIERARCHICAL PROPERTY LOADER and DEFAULT PROPERTIES BUNDLING, you can use externalized default properties.

Known Uses

The Java platform [4] contains a class `Properties` with the method `getProperty(key, defaultValue)` whose semantics are as described by this pattern.

Property File Templates

Context

An application uses a PROPERTY LOADER and stores properties in property files.

Problem

When configuration information is externalized in a property storage, the source code and the actual properties available in a specific deployment environment may not be consistent any more. Errors may arise when users or administrators manually change the content of the property files.

How do you keep externalized properties consistent with the application code?

Forces

The following *constraints* need to be resolved:

- *Maintenance*: Property files can be copied and modified freely. It can be difficult to find out and maintain the correct version of a property file for a specific version of an application.

- *Robustness*: An application that reads a property file with properties that are inconsistent with its own version (different property keys or allowed property values) may not work properly.

- *Updating*: A new version of the application may introduce new properties that must be made available in every installation after the upgrade.

Solution

Include PROPERTY FILE TEMPLATES into the build process so that they are bundled with the application binaries and unbundle these files as part of the installation process so that they can then be modified by the administrator or user.

A PROPERTY FILE TEMPLATE contains all properties that the application relies on. Every property should have a default value that enables the application to work.

When a new version of an application is installed, the new PROPERTY FILE TEMPLATE is also installed, possibly overwriting an existing, earlier version of the same file. The administrator of the installation (or a user) then needs to adapt the properties as needed.

Consequences

PROPERTY FILE TEMPLATES offer the following *advantages*:

- When properties are changed (e.g. as part of a system refactoring) and new binaries are deployed, these changes are deployed in all updated environments at once without further manual effort.

- The property files can be stored in the same repository as the source code so that the history of changes to the application is synchronized with the history of changes to the property files.

The solution also causes the following *liabilities*:

- For some properties, e.g. database passwords, there are no defaults available to put into a Property File Template or it is not feasible to put secret values into a file that is available to all developers. With a Hierarchical Property Loader and Default Properties Bundling you can keep such information outside the bundled property files.

- When doing an upgrade, existing property files are overwritten. Therefore, manual effort and great care is necessary to restore all property values that had been modified before the upgrade.

Known Uses

Unix packaging systems such as Debian [6] or RPM [7] bundle application binaries with default settings that can subsequently be modified by system administrators.

Mac OS X applications are distributed as folders having both binaries and property files at well-known locations. These bundle packages [8] are presented to the user as unique items.

Programs written for the Java platform usually distribute binary code in the form of JAR files (ZIP format) that can also bundle property files with application logic [9]. An example of this is the Maven build system which stores build meta information as properties in a JAR file [12].

Hierarchical Property Loader

Context

An application externalizes properties from the source code by using a PROPERTY LOADER that loads property values at start-up time and provides an API to retrieve the values at runtime.

Problem

An application may need to be configured differently in different environments, for different users, or on different hosts. Maintaining separate property storages for every possible combination of these factors may be very expensive, in particular because, quite often, only a few property values differ between the environments.

How can you provide an application with customized, i.e. environment-specific or user-specific, configuration settings without maintaining separate, mostly redundant property storages?

Forces

Several *constraints* and *requirements* make the problem difficult:

- *Maintainability*. Redundancies among the property storages should be avoided. If a default value exists for a property, that value should be declared only once. If the value of a property needs to be changed for a specific host or user, only that property should be redefined.

- *Administrability*: It should be easy for a system administrator to recognize which properties still have default values and which property values have already been changed locally.

- *Compatibility*. It should be possible to introduce new configuration properties without the need to manually modify existing local property storages when an installation is updated.

- *Security:* In most enterprise setups, developers should not have access to some properties such as production server passwords. Yet, it should still be possible for developers to provide defaults for non-critical properties.

Solution

Extend the PROPERTY LOADER to handle not only a single property storage but multiple storages and let it process the application properties from these storages in a way so that property values from a more specific storage overwrite the values from a less specific storage.

Create a default property storage with all properties that the application relies on and assign a default value to each property. For each supported environment, host, or user create an additional property storage that contains only those properties whose values differ from the defaults.

When the application starts up, let the HIERARCHICAL PROPERTY LOADER determine the current environment, host, and user and check for the respective property storages. Let the loader read first all default properties and then all properties from the more specific

property storages: every property value of a more specific property storage overwrites the values of the less specific property storages (see illustration 2).
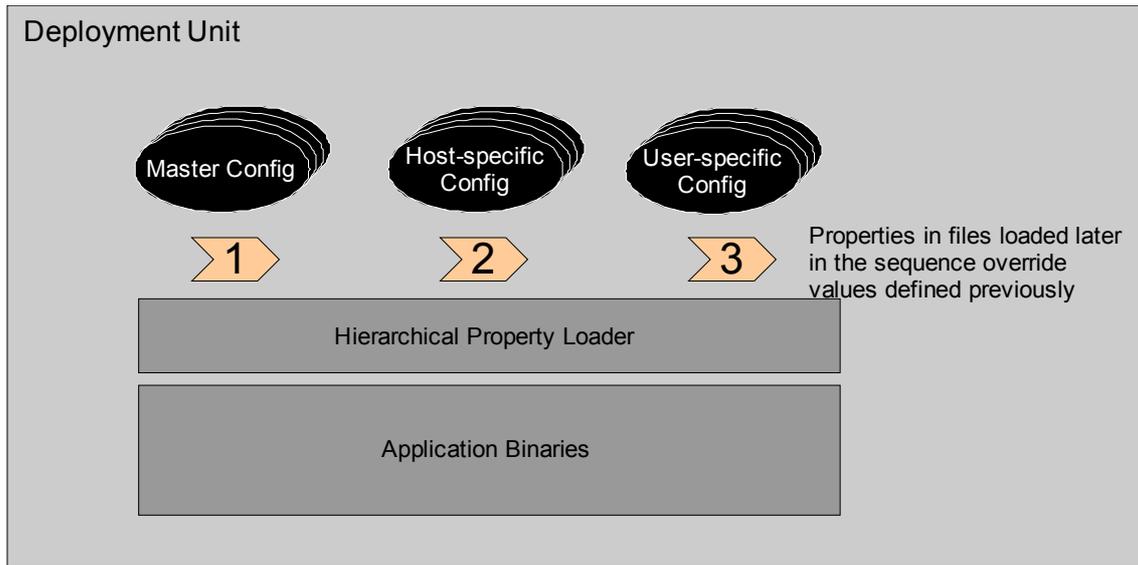
Illustration 2: Structure of deployment unit with a HIERARCHICAL PROPERTY LOADER

Security-critical properties are stored in property storages that are only available locally whereas non-critical properties may be stored in default property storages. Security is enforced by means of the local platform. It is possible, for example, to set the permissions of a local properties file so that only the administrators and the application itself may read these properties.

In a complex scenario, a single property may be overwritten by values from several property storages. In order to make the actual value of a property at runtime comprehensible, the HIERARCHICAL PROPERTY LOADER internally stores information about the origin and overwritten values of each property.

If property files are used as property storages, you could implement the pattern by creating separate property files for each environment, host or user and apply a naming convention to make the HIERARCHICAL PROPERTY LOADER aware of the appropriate files. For example, you could call the master property file `config.properties` and environment, host or user specific property files `config.<environmentname>.properties`, `config.<hostname>.properties` or `config.<username>.properties`, respectively.

If a database table is used as property storage, consider adding a database column as qualifier that takes the name of the environment, host or user to which a property entry belongs. By filtering property entries by this qualifier you could retrieve the respective properties by executing several queries and handling the retrieved properties as described above.

An typical application is able to easily determine the name of the host it is deployed on and the name of the user that started the application. The name of the environment the application is currently deployed in (i.e. test, integration, production), is typically not available by standard means. Instead, quite often, a proprietary system variable is declared outside the scope of the application itself and queried from within the application. Such a system variable could be set, for example, in a system start-up script or in the application's start-up script.

Consequences

The HIERARCHICAL PROPERTY LOADER offers several *advantages* over the normal PROPERTY LOADER:

- Redundancies among property storages are avoided because environment-specific property storages don't duplicate default values.

- You can bundle both default and environment-specific properties with the binaries and still allow a system administrator or user to change some property values in a local, not-bundled property storage.

- The unique properties of an environment are clearly documented by the content of environment-specific property storages because these storages contain only changes to the default settings.

- The HIERARCHICAL PROPERTY LOADER is responsible to determine the current environment and to provide access to the configuration properties. This encapsulation reduces the possibility of errors when retrieving configuration properties at runtime.

- Developers can change properties in property storages of their own that only affect their development environments without the danger of unintentionally checking in any changed default properties to version control systems.

The HIERARCHICAL PROPERTY LOADER also causes *liabilities* of its own:

- The actual complete configuration settings of a system in a specific environment cannot be determined any more from a single source.

- The more hierarchies the loader has to consider, the more complex the property handling becomes, in particular for a system administrator that needs to know how to determine and change properties quickly.

- The application needs a robust mechanism to identify the current environment, which might be difficult if the application is deployed on a cluster or in a virtualized environment. If, for example, the name of host on which the system is deployed is used as qualifier, the system cannot be easily moved to another host any more.

- If several applications are deployed and run on a single machine or cluster, you must take care that the applications' property storages are not mixed up.

- Because every property needs a default value, you cannot enforce that some properties for which there are no valid defaults need to be defined in environment specific property storages. PROPERTY ENFORCEMENT helps in this case.

Variants

You can change the number of hierarchy levels, which allows for more fine-grained control of environments. For example, users might have different database account names in different environments. In such a setting, the mechanism described above is not sufficient. Extending the hierarchy levels is straightforward – in case of property files, adding a new property file `config.<hostname>.<username>.properties` that overrides all other files does the job.

Besides a distinction in environment, host, and user, other distinctions are possible as well, for example for language and country specific settings.

Known Uses

In standard UNIX systems, hierarchical properties are a common pattern. For many programs, first a system-wide configuration is read, then properties are replaced by user-specific values or even from arguments given on the command line. A well-known application using this pattern is OpenSSH. It allows for global configuration using `/etc/ssh_config`, user-specific configuration with file `~/.ssh/config` and for command-line options via the parameter `-o <option>` [2].

Microsoft Group Policy applies a similar approach to setting policies for Windows machines. Group policies define defaults, user policies get into more specific details [3].

Default Properties Bundling

Context

A Hierachical roperty Loader uses property files as property storage.

Problem

Once configuration information is externalized and distributed over multiple configuration files, the source code and the actual properties available in a deployment environment may not be consistent any more.

How do you keep externalized properties that are hierarchically evaluated consistent with the application code?

Forces

The following *constraints* need to be resolved:

- *Robustness*: An application may depend on the availability of specific property values.

- *Updates*: When an application update is installed, new property values must be made available in the local installation.

- *Administration*: If an application is installed only in a few different environments (e.g. one test, one integration, and one production system), the respective property settings for each environment may be known at development-time and should be maintained at a single location.

Solution

Include the property files for all supported environments into the build process so that they are bundled with the application binaries. Do not unbundle them as part of the installation process so that they stay unchanged after the installation.

First, include the property file that contains global default properties so that for all properties that the application relies on property values are available. Also include property files with properties for specific environments whose values are well-known at development time.

By using the Hierarchical Property Loader, all property values can still be overwritten by values from local property files.

Consequences

Default Properties Bundling offers the following *advantages*:

- When properties are changed (e.g. as part of a system refactoring) and a new version of the application is deployed, the changed properties affect all updated environments at once without further manual effort.

- Default values for specific environments can be changed and new properties can be introduced without affecting an installation's manually set properties.

- Default property files can be stored in the same repository as the source code so that the history of changes to the application is synchronized with the history of changes to the property files.

The solution also causes the following *liabilities*:

- When application developers change the values of default properties and these properties are not overwritten locally, the behavior of an updated application might change without being noticed by the administrators.

- Although default values can be set and deployed for every property, it is not possible to enforce setting properties locally that can not or must not be deployed (e.g. passwords). Consider using PROPERTY ENFORCEMENT then.

Known Uses

Many Unix applications deliver default properties in their packaged application binaries and allow the creation of a user-specific property file where all properties can be changed.

The Windows Registry [5] allows applications to write properties into the system part during its installation and to let users modify properties in the user part of the registry.

Property Enforcement

Context

A HIERARCHICAL PROPERTY LOADER reads from property storages for specific environments, hosts, or users and provides an API to retrieve the values at runtime.

Problem

When a property may be set and refined in multiple property storages, it may happen that a property value is not set at all for a specific environment.

How can you ensure that a mandatory property for which no default value exists is set in an environment, host, or user specific property storage?

Forces

The following *constraints* apply:

- *Fail early*: If a property is mandatory but no default value exists, the application should fail early, i.e. before the property is requested for the first time.

- *Notify user*: If the application fails due to a missing property, the application user should be told how to fix the problem.

Solution

Add all mandatory properties for which no default values exist to the default property storage and assign a dedicated value to them so that the HIERARCHICAL PROPERTY LOADER can check whether this value has been changed in a more specific storage.

In the default property storage, set the property value of each mandatory property that does not have a default value to a value such as "TO_BE_DEFINED". After the HIERARCHICAL PROPERTY LOADER has loaded the properties from all property storages, let the loader check whether any property still has the "TO_BE_DEFINED" value. In that case, let it clearly state the misconfiguration and, if no other option is suitable, let it terminate the application.

Consequences

The given solution has the following *advantages*:

- By marking all properties in the default property storage as described, mandatory properties without defaults are clearly documented.

- If a software upgrade introduces a new mandatory property, there is no risk that the system administrators might miss setting the appropriate value.

The solution also has a *disadvantages*:

- Care has to be taken to spell the default value correctly – a slight misspelling can cause the system to not notice missing properties.

Known Uses

The Confluence Wiki requires administrators to set the confluence.home property [14].

Recursive Property Resolution

Context

A (HIERARCHICAL) PROPERTY LOADER reads from property storages and provides an API to retrieve the values at runtime.

Problem

Sometimes, a set of properties needs to be changed at once because the properties are closely semantically associated and it would be wrong to change only individual properties. For example, there are several databases to connect to and a couple of properties such as the host name, user name, and password need to be changed to connect to each one.

How do you create a connection between individual properties so that if their values need to be changed all of them are changed at once?

Forces

The problem shows the following *forces*:

- *Consistency*: If some properties must always be changed together, it is important to make these changes consistently.

- *Maintainability*: It must be clear which properties are connected.

Solution

Extend the PROPERTY LOADER by adding the ability to resolve indirections between properties so that by changing the value of one property, the values of a set of properties that refer to the changed property are changed as well.

Extend the PROPERTY LOADER so that properties can not only be mapped to concrete values but also to other properties which are resolved subsequently. Consider the following example:

```
environment=dev

dev.dataSource.username=devuser
dev.dataSource.password=devpw
ci.dataSource.username=ciuser
ci.dataSource.password=cipw

dataSource.username=${${environment}.dataSource.username}
dataSource.password=${${environment}.dataSource.password}
```

In this example the property `dataSource.username` is first resolved to `${dev.dataSource.username}` and then to `devuser` and, accordingly, `dataSource.password` is resolved to `devpw`. If the value of the property `environment` would be changed to `ci`, the value of `dataSource.username` would therefore be resolved to `ciuser` and the value of `dataSource.password` to `cipw`.

Consequences

The solution has the following *advantages*:

- By changing a single property value, you can switch between sets of property values.

- If a HIERARCHICAL PROPERTY LOADER is in use, all foreseeable possible values of property sets may be added to the default property storage. Only the value of the "switching" property needs to be declared in a more specific property storage.

The solution also has some *disadvantages*:

- Recursive property resolving adds another layer of indirection that makes it more difficult to understand which properties are actually in use by a system, in particular if the affected properties are spread across multiple property storages.

Known Uses

The Apache Commons Configuration framework allows for recursive property resolution [15] very similar to the pattern described above. The Rails programming language supports different property settings for three testing environment settings that effectively work as a recursive property resolution – based on a single switch property (RAILS_ENV), other properties are adjusted accordingly [16]. The Spring Framework [17] contains a class `PropertyPlaceholderConfigurer` that, since version 2.5 of the framework, is able to recursively resolve properties.

Dynamic Property Loader

Context

A (HIERARCHICAL) PROPERTY LOADER reads from property storages and provides an API to retrieve the values at runtime.

Problem

The PROPERTY LOADER loads properties at start-up time and stores them internally. Properties are not checked every time they are accessed for performance reasons: Some storages may impose relatively long waiting times when being queried. Still, many applications have very limited time slots for maintenance and restarting.

In other cases, it is desirable to change settings back and forth while the application runs, for example to temporarily change the logging level at a production system.

How can the values of some properties be changed without restarting the application?

Forces

The following *forces* apply:

- *Uptime*: Restarting an application to reload the properties from their property storages might not be feasible for some applications to comply with quality of service agreements. In particular, server applications might take a considerably long time to start-up.

- *Performance*: Switching property values at runtime needs to be fast to avoid performance problems, in particular in large sites under load.

Solution

Extend the (HIERARCHICAL) PROPERTY LOADER so that it reloads the properties from all property storages on request.

Special care must be taken to prevent the access to properties while they are reloaded and to remove references to properties values that might not be valid any more after the reload.

Implement any of the following three strategies to trigger the reload: (1) The PROPERTY LOADER may check for changed properties on every access to property values, (2) may periodically scan for changes to local property storages and/or (3) may provide an API to be explicitly notified when property storages change and a reload must occur.

From a performance point of view it is usually not feasible to execute full hierarchical property loading each time a property is accessed. Some cache invalidation strategy is necessary. With implicit invalidation, a periodic watchdog scans all property storage candidates for changes and reloads properties if necessary. With explicit invalidation, the re-evaluation of the hierarchical property loading is triggered via some signal (e.g. a JMX command [10]).

The choice between implicit and explicit cache invalidation is a tradeoff between quick round-trip times and prevention of accidental reconfiguration. For example, hierarchical

property loading for logging configuration benefits a lot if any changes are loaded automatically – the potential damage of a misconfiguration is usually low.

Consequences

Dynamic property loading has the following *advantages*:

- Properties can be changed without restarting an application.

- It is also possible to trigger the reload of property values in a cluster of application servers nearly at the same time if the servers share the same base property storage, e.g. property files from a shared directory via NFS.

The solution also has some *disadvantages*:

- Changing properties may cause inconsistent system behavior if some operations assume constant property values over a period of time.

- Administrators must be aware that they must not change property storages while the application is running if the change should not have an immediate effect.

- Implementing a strategy to detect that properties need to be reloaded may harm the application's performance.

Known Uses

The Apache Web Server can be told via a control command (`apachectl graceful`) to re-scan its configuration files without effectively restarting [18].

The Jetty servlet engine constantly monitors its context directory and reconfigures itself by restarting only specific web applications if any changes occur [19].

Most database systems maintain parts of their own configuration as internal database tables; as such, the configuration properties can be changed at runtime using standard SQL commands. An example is MySQL [20].

The Log4J Java Logging framework implements a periodic check of changing configuration properties via its `PropertyConfigurer#configureAndWatch()` method [11].

Pattern Language Example

The patterns described in this paper build upon each other and are usually combined. In this section, we give an example combining the PROPERTY LOADER, HIERARCHICAL PROPERTY LOADER, RECURSIVE PROPERTY RESOLUTION and PROPERTY ENFORCEMENT patterns.

The Java platform offers a standard API to load properties [4]. As such, it serves as the basis of our example. Yet, we have also implemented the pseudo code described below in the Ant build system. Adoptions to other platforms are straightforward.

We assume that the default configuration is declared in a file called `config.properties`. It is stored with bundled with the application binary, as well as the host-specific configuration file `config.earth.properties` and the user-specific file `config.joe.properties`. On the deployment environment, a file `config.dev.properties` overrides specific parameters.

```
// Property Loader pattern
Properties props = read("config.properties")        // default properties

// Hierarchical Property Loader pattern
String env = System.getEnv()                        // assume "dev"
String host = System.getHostname()                  // assume "earth"
String developer = System.getCurrentUser()          // assume "joe"

File[] overrides =
    ["config." + env + ".properties",               // config.dev.props
     "config." + host + ".properties",              // config.earth.props
     "config." + developer + ".properties"]         // config.joe.props

foreach file in overrides do
    Properties newProps = read(file)
    foreach newProp in newProps do
        props.setValue(newProp.name, newProp.value)
    done
done

// Recursive Property Resolution pattern
while (props.containsPropValueContainingOtherProp)
    String oldValue = propWithValueContainingOtherProp.value
    String newValue =
        oldValue.replace("${" + otherProp.name + "}", otherprop.value)
    propWithValueContainingOtherProp.setValue(newValue)
done

// Property Enforcement pattern
String UNDEFINED = "<HAS_TO_BE_DEFINED>"
if props.containsValue(UNDEFINED)
    throw Exception ("Some required properties are not set")
```

Acknowledgements

References

All WWW links are valid as of January 10th, 2010.

[1]   Apache Ant Project - http://ant.apache.org/

[2]   OpenSSH manual - http://www.openssh.com/manual.html

[3]   Jakob H. Heidelberg, Managing Windows Vista Group Policy (Part 2) -
        http://www.windowsecurity.com/articles/Managing-Windows-Vista-Group-
        Policy-Part2.html

[4]   The Java Tutorials: Properties –
        http://java.sun.com/docs/books/tutorial/essential/environment/properties.html

[5]   MSDN – Windows Registry –
        http://msdn.microsoft.com/en-us/library/ms724871(VS.85).aspx

[6]   The Debian GNU/Linux FAQ; Chapter 7. Basics of the Debian package
        management system – http://www.debian.org/doc/FAQ/ch-pkg_basics.en.html

[7]   Maximum RPM: Taking the Red Hat Package Manager to the Limit; Chapter 13.
        Inside the Spec File –
        http://www.rpm.org/max-rpm/s1-rpm-inside-files-list-directives.html

[8]   Apple Developer Connection: Bundle Programming Guide – http://developer-
        .apple.com/mac/library/documentation/CoreFoundation/Conceptual/CF-
        Bundles/index.html

[9]   The Java Tutorials: Packaging Programs in JAR files –
        http://java.sun.com/docs/books/tutorial/deployment/jar/index.html

[10]   The Java Tutorials: Java Management Extensions (JMX) –
        http://java.sun.com/docs/books/tutorial/jmx/index.html

[11]   Log4J Manual - http://logging.apache.org/log4j/1.2/manual.html

[12]   MaestroDev, Better Builds with Maven, Chapter 5.4.1 -
        http://www.maestrodev.com/better-build-maven

[13]   Configuring Logging in Atlassian Confluence -
        http://confluence.atlassian.com/display/DOC/Configuring+Logging

[14]   Installing   Confluence   EAR-WAR   on   Tomcat   -
        http://confluence.atlassian.com/display/DOC/Installing+Confluence+EAR-
        WAR+on+Tomcat

[15]   Apache Commons Configuration, PropertiesConfiguration class -
        http://commons.apache.org/configuration/apidocs/org/apache/commons/config-
        uration/PropertiesConfiguration.html

[16]   Rails testing environments - http://guides.rubyonrails.org/testing.html

[17]   Spring Framework - http://www.springsource.org/about

[18]   Apache Web Server apachectl documentation -
        http://httpd.apache.org/docs/2.2/programs/apachectl.html

[19]   Jetty Servlet Engine Context Deployer -
        http://docs.codehaus.org/display/JETTY/ContextDeployer

[20]   MySQL Server Administration Documentation -
        http://dev.mysql.com/doc/refman/5.5/en/server-administration.html