UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science
Information Technology

Ats Uiboupin

Decoupling environment configuration from application archive

Master's Thesis (30 ECTS)

Supervisors: Neeme Praks
Helle Hein, PhD

Author: ..... "....." May 2013
Supervisor: ..... "20" May 2013
Supervisor: ..... "20" May 2013

Allowed to defence
Professor: : ..... "....." May 2013

TARTU 2013

Contents

Introduction

Every application that aims to be reusable in different environments, needs a mechanism that can be used to receive the environment-specific configuration. Therefore deploying a web application often requires much more than just packaging and uploading it to a server. Most of the time different configuration is needed for running the application in development, testing and production environments.

One example of the configuration that is usually different for all environments is connection data for database server: host and port of the database server, username and password. Another feature that may need to be configured based on the environment is sending emails. In development and test environments it may be desired that organisation's internal mail server is used that could send emails only to organisation's own email addresses – that also requires defining different settings based on the environment where the application is deployed.

One primitive way that has been used to configure small applications is hard-coding all the configuration data into the application. However, this approach has many disadvantages. Mainly, when the configuration needs to be changed even for the same environment, for example because it was decided that the application should send out emails from another email address, then the application code would need to be changed. A research claims that $15000 was wasted to migrate the application with hard-coded configuration to the new hardware [1].

A better solution is externalizing the configuration into configuration files. But even in this situation the application is often rebuilt from scratch to package it into some kind of archive that could be manually delivered to the environment where it should be used. Depending on the simplicity and level of bureaucracy required for the delivery process this often involves several people, such as delivery manager, project manager and software developer. Evidently, the more needs to be done and the more people need to be involved, the more time and resources it will take.

It makes sense that when configuration value needs to be changed, then the whole application should not need to be rebuilt from scratch, but instead a single configuration parameter should be modified. Changing configuration parameters is often sole responsibility of delivery manager and it should not require involving other people, such as developers, to do a simple configuration change.

The purpose of this study is to investigate better solutions to decouple environment-specific configuration from application package to allow easily changing the configuration.

The first Chapter of the present study gives a small overview of different configuration methods (1.2) and how they could be used to configure the application to be deployed to different environments that require different settings (1.3). It also gives a brief overview of an application deployment tool called LiveRebel [2] that was improved with an environment-specific configuration management feature developed as a

part of this research. The second Chapter describes the features and architecture of the environment-specific configuration management solution implemented for LiveRebel.

Chapter 1

Background

There are several different ways to set configuration for the application, but usually several approaches are combined for the best results. The following Sections will cover different basic configuration methods without focusing on how they could be used to configure the same application for different deployment environments. Before covering different configuration methods, the next Section will briefly explain some terms that are used in this research.

1.1 Terminology

This study mentions several terms that are not widely adopted or may be used to refer to different concepts by different software or studies.

One such term is **deployment environment** – usually it is used to refer to a group of servers that work together to provide the same service. Deployment environment could contain different types of servers, such as database server, web server and load balancer. Often deployment environment consists of many servers of the same type to provide better performance or a backup solution in case of failure of the primary server. Usually servers of the same type in the deployment environment also have similar configuration. Often the deployment environment is just referred to as "environment". For example, production environment, quality assurance environment, performance testing environment or development environment.

Configuration parameters and **configuration properties** are used in this dissertation interchangeably, but the same concept is kept in mind. Both of them are essentially a set of key and value pairs that can be used to configure an application.

Application package and **application archive** both refer to the same thing – one file containing the application code and potentially resources and configuration for the application.

1.2 Configuration methods

This Section introduces basic configuration methods that could be used to configure the application. Often these methods are combined to avoid disadvantages of the individual configuration methods and to benefit from the advantages of those methods. However, this Section focuses on the individual methods, not how to combine them.

1.2.1 Configuration via command-line arguments

Standalone programs, such as command-line programs or desktop applications, usually accept startup arguments that are used as an input for the program that affects the output or behaviour of the program. For command-line programs these arguments are usually called command-line arguments. Configuration can also be considered as specific sort of input to the program and hence command-line arguments could be used to configure the program. This means that the application that can be started using executable file `ExampleProgram.exe` could accept different arguments that affect the behaviour of the program. That program could be started using command-line example from Listing 1.1 to enable logging debugging statements in test environment that should be written to a file specified with the second argument:

Listing 1.1: Example of setting configuration through command-line arguments.

```
ExampleProgram.exe logLevel=DEBUG logFile=c:\tmp\myProgram-
    debug.log
```

This approach is one of the easiest to implement in some cases. However, it may be tedious to use due to the fact that each time an application is started, configuration parameters need to be typed in again. This can be worked around by creating shortcuts on Windows operating systems or scripts on UNIX-like operating systems and if the program is always started the same way after initial configuration, then it may be one of the easiest solutions.

On the other hand, problems could arise if the total length of command-line including parameters could exceed certain amount of characters, because some operating systems have limitations on the length of the command-line [3]. This means that you may be unable to pass all the needed input to the program. In addition, this approach is not convenient for the users and setting values each time from command line is error prone. Also configuration parameter values cannot be changed after application is started.

Another disadvantage of this method is that it is not practical to use it for configuring a web application. Web server is one program that itself could use command line arguments, but all web applications are served through the web server program. Since individual web applications are not started using command line, they cannot be individually configured using command-line arguments.

1.2.2 Configuration from the server environment variables

Applications written in many programming languages can detect several things about the surrounding environment where a program is currently running. For example, a program can detect, using different means, operating system, computer name and even hardware installed on the computer.

One simple method that can be used by the application to detect some things about the surrounding environment, where the program is currently running, is environment variables [4]. Some environment variables are set by the operating system by default, but users and applications running in the operating system can also set and edit environment variables. During the start of the program, the application can inspect the environment variables and use them the same way, as it could use command-line arguments.

When comparing the usage of environment variables to the usage of command-line arguments, then there are some advantages with this approach. After setting the environment variable, it can be used by the application every time it is started from the same environment. It means that starting the application with the same configuration is easier after initial setup of the environment variables – it is not needed to type the same configuration parameter values each time when the program is started. It also eliminates potential mistakes that can happen when typing the same configuration, as arguments to command-line, every time before starting the application.

On the other hand, one disadvantage of this approach is that environment variables are available to all programs started from the same environment. If all applications would configure themselves only via environment variables, then it would be too hard to understand what environment variables are actually used by some specific application – all applications could use any environment variables even if some kind of variable naming convention would be used by some applications.

1.2.3 Configuration from file

Another simple way of defining the configuration for the application is via configuration file. Configuration file could be located by the application using absolute path in file system, relative path from the application or using path to file inside application archive.

Standalone applications often use relative path from the application entry point to allow installing application to any folder chosen during the installation. At the same time it allows storing application configuration under the application installation directory. Web applications often bundle some of its configuration inside web application archive that could be read at runtime.

There are several benefits for this configuration method when comparing it to the configuration method that passes all the configuration parameters through the command-line arguments. One benefit is that the configuration of the program does not need to be typed in for every startup of the application. Often previous configuration can be reused without any changes or just a few small changes could be made without changing the rest of the configuration stored in that file. It also allows to use different file formats for expressing the configuration - for example, if the configuration data has a hierarchical nature, it might be more convenient to store it as XML or JSON. Additionally, the length of the configuration file content is only limited by the maximum file size that the operating system supports.

One negative aspect of using configuration files is that if the same configuration must be used by the application running in many servers, then configuration files on each server must be updated instead of updating configuration in only one central place.

1.2.4 Configuration using configuration service

One of the advanced options is to use some kind of configuration service. With this approach the application is asking configuration parameter values at runtime in contrast to configuration provided before startup. This method could also be used to reconfigure the application after configuration parameter values are updated. The application should be notified about configuration changes and then it can use new configuration

values to update its configuration.

When the application needs to use configuration parameter value, it will contact the configuration service. Each server could have its own local configuration service that is used by applications running on that server. The benefit of using local configuration service is that each server could have its own configuration settings. But when the same application must be deployed to several servers with the same configuration, then configuration would be duplicated in each local configuration service.

To avoid duplicating the same configuration for all the servers that should receive same configuration, remote configuration service could be used. That way many servers could reuse the same configuration that is managed centrally.

Another approach to avoid duplicating the same configuration values for many servers that must be configured similarly, would be to combine local configuration and remote configuration service features. Local configuration service could use remote configuration service to detect configuration parameter values that are not configured on local service. In this case common configuration values could be provided by remote configuration service and local configuration service could override those values when needed.

1.2.4.1 Existing solutions

Database used by the application could be set up the way that it could also contain configuration for the application that is using it primarily to store data. With this setup even a database can be considered to be a primitive configuration service that can be used by all servers running the application in the same deployment environment. However, configuration for the database itself needs to be configured for the application using other configuration methods.

On Java platform Java Naming and Directory Interface (JNDI) could be used as a local configuration service [5]. JNDI itself is not a specific solution for configuration management, but as the name suggests, it is a general solution for naming and directory services. What responses JNDI service will generate for a specific query, depends totally on the configured implementations of JNDI Service Provider Interface (SPI), but there has to exist at least one SPI implementation. If one JNDI SPI implementation would provide configuration values, then it could be used as a configuration service.

There exist solutions for other development platforms as well. One example of specialised configuration management service is Escape [6]. Escape is configuration management server written in Ruby, but server can also run on JVM and it is easy to use from Java and Python, due to existing client libraries that internally use REST API exposed by the Escape configuration management server [7].

When compared to previously mentioned configuration methods, it takes more effort to set up configuration service and start using it from the application. On the other hand, separate configuration service adds flexibility to the application configuration – configuration for applications running on different servers can be managed centrally.

1.2.5 Exposing configuration interface

There is one more configuration strategy that could be used to change the behaviour of the application. However, this configuration strategy is fundamentally different from all the previous strategies. If an application can be deployed and started with default configuration values (or configuration values provided through other means)

and there may be situations where the default configuration may need to be changed, then the application itself could provide configuration interface that can be used to change the configuration at runtime. It could be implemented as a settings page of the application, or a configuration service might be exposed to other tools that could be used to remotely reconfigure the application.

On Java platform Java Management Extensions (JMX) technology provides standard way to expose reconfigurable resources of the application [8]. JMX can be used for building managing and monitoring features through standard infrastructure. JMX is even used to monitor and manage the Java Virtual Machine (JVM) itself that is used to run all Java applications.

Summary of configuration methods

This Section introduced several basic configuration methods that can be used to configure applications. Command-line arguments (1.2.1) and environment variables (1.2.2) are often used to pass couple of arguments to the application being started. Usually configuration files (1.2.3) are used instead of command-line arguments and environment variables when application uses many configuration parameters that can be reused when starting application next time.

In addition to the basic configuration approaches, this section also introduced more advanced methods that can be used to change the configuration of the application even while it is running (1.2.3, 1.2.5, 1.2.4 and 1.2.5), without requiring to restart the application after changing the configuration. An application that exposes configuration interface (1.2.5) can provide means to change configuration of the application without restarting it. Using configuration service (1.2.4) is useful when it is desired to centrally configure (and reconfigure without restarting) applications running on different servers.

However, when it is desired to deploy the same application to several deployment environments with different configuration, then these configuration methods are typically combined to provide more flexibility for managing configuration. The following Section will cover ways that can be used to configure application more conveniently for specific deployment environment when there are several different environments where the same application should be deployed.

1.3 Configuring application for different environments

There are several options to implement environment-specific configuration solutions, when it is known that the same application will be deployed to several different environments, such as for example quality assurance, performance testing and production environments. This Section covers different approaches for configuring the same application for each different environment.

1.3.1 One application bundle with multiple configurations

One option is to create a single application archive that bundles configuration files for all environments where this application might be deployed. For example, there could be a configuration folder that contains one folder for each environment configuration. By knowing only the name of the configuration folder that should be made available

to the application by other configuration methods, the application will know where it needs to load the configuration from.

One benefit of this approach is simplicity. First of all there is a single application archive for all known environments. The second benefit is that running application needs to receive only one configuration parameter from outside the archive – the name of the configuration. As it was discussed in the previous Section, there are several easy ways to pass a single configuration parameter to the application.

There is also one major disadvantage of this approach. Every time when configuration needs to be changed for some environment, a new application archive must be built. When the configuration of the application is versioned with source code of the application and assembled during build process, then it means that at first the configuration must be changed and after that build process started. Depending on the complexity and workflow used in the delivery process this often involves several people, such as delivery manager, project manager and software developer – for that reason it may take a lot of time for simple configuration change.

1.3.2 Multiple application bundles

Another way to configure an application for different environments is to create several environment-specific application bundles, each containing application code and configuration, which is specific for one environment.

One reason why this approach could be chosen is to reduce the potential for security breaches between environments. For example, production environment database passwords could be included in the configuration. If production environment configuration is in the same application archive as all the configurations for other environments, then more people could potentially cause security issues for that environment.

1.3.3 Semi-built application archive

Another approach is to divide building process into steps. In the first step, an intermediate application archive without environment-specific configuration would be built. The second step would post-process the archive that takes configuration for particular environment and intermediate application archive produced by the first step and merges them into a final application archive. The produced application archive could be deployed to environment corresponding to the configuration that was used to produce the application archive in the second build step.

Adding configuration in separate build step could be just a small optimization that reuses single intermediate build result for creating final archive in the next step for all environments. However, application deployment tools could also benefit from this approach by automatically injecting environment-specific configuration before deploying – configuration could be chosen based on the server where the application is chosen to be deployed to.

1.3.4 Using installers to prepare server environment for application deployment

For an organisation that need to run several instances of the same application with the same configuration in different machines it may make sense to create an installer

that prepares the computer for a specific application version running in a specific environment.

In addition to downloading dependency programs and environment-specific configuration files to places where application expects them to be, installer could also update some existing files, for example hosts file or even set environment variables. This process is called *provisioning* the server for the application being deployed.

UNIX-like operating systems have had for more than a decade tools for downloading and installing software by executing single command-line. One such tool that can also be used for installing environment-specific configuration for the server is Debian's Advanced Package Tool (APT) [9]. Free provisioning tools like Chef [10] and Puppet [11] that have emerged lately, provide even more flexibility and they can be used on Windows operating system as well. These tools can be used for provisioning a server with needed software that are configured exactly the way that the application being deployed there is expecting. Provisioning tools could download or update configuration files that are used by the application, so that the application itself does not need to be tweaked at all for the environment.

One option for configuring server for specific environment is to create different installation packages for each environment with different name. This means that environment-specific configuration package must be chosen when preparing a server for a specific environment.

This approach can work well for some organisations, but for others it may be too much effort for simple and small environment-related configuration changes. It may take too much time to create and maintain the installer over time for each environment.

Another problem with this approach is that application code and application configuration is totally separated from each other. It could happen that the server is prepared for a wrong version of the application and the application cannot run properly – like one version of the application trying to use relational database that has an incompatible schema.

1.3.5 Configuring application at runtime

Sometimes the configuration of the application may need to be updated when the application is already running. For those cases several approaches mentioned in previous Subsections of this Section, cannot be used. As discussed before, command-line arguments and environment variables cannot be changed after starting the program and changing files embedded inside a deployed application archive may be impossible on some programming platforms.

On Java platform JMX could be used to change configuration values at runtime. JMX is technology that supplies tools for managing and monitoring applications, but it is not a specialized tool for environment-specific configuration management. It provides means that can be used to connect to a running program and change some property if needed, but that may not be the most convenient solution for setting an environment-specific configuration. It can be used to reconfigure the application, but since the application itself cannot ask the environment-specific configuration, it should either (*i*) wait until it is configured through JMX before completing startup process, or (*ii*) receive default configuration via other means.

As mentioned before, one example of specialised configuration management service is Escape. It allows central configuration management for all environments and for all

applications. Listing 1.2 is an example of how all configuration parameters could be retrieved through Escape Java client API [12] from Escape server for a given application running in a given environment. Last line reads configuration parameter value based on parameter name.

Listing 1.2: Code example for getting configuration parameter value from Escape configuration server.

```
Properties properties
  = Client.getProperties(serverUrl, envName, appName);
String paramValue = properties.getProperty(paramName);
```

One benefit of this approach is that environment-specific configuration for all the environments can be managed centrally from one place. However, a small disadvantage is that running application has to know the configuration server url and the name of the environment it is currently running – these two parameters must be received through other configuration methods. The name of the application must also be passed to the configuration server, but this is something that could be hard-coded into the application.

There are several things that have to be considered while building an application that configures itself from a remote configuration server.

Firstly, the performance of the application must be kept in mind. Reading configuration from remote computer may be much more resource consuming than loading it from a local file. Secondly, the performance of the central configuration server must also be kept in mind. If it serves configuration for hundreds of applications in several environments, and applications do not cache configuration values, then configuration server may be unable to serve all those requests in reasonable time.

Another thing to keep in mind is that configuration server must work when the application asks its configuration. Often the application just asks configuration from the configuration server just once during the startup, but when the application is expected to be reconfigurable, then the application may ask configuration values from the configuration server at any moment in time. This may mean that the application needs to be prepared for situations when the configuration server is unreachable for whatever reason. The configuration server could become unreachable because of network or server's hardware problems and also for example when the configuration server is taken down for maintenance.

Summary of the Section

There are three approaches, where environment-specific configuration could be stored for the application.

1. Subsection1.3.1, 1.3.2 and 1.3.3 discussed methods that embed environment-specific configuration in the application.

2. Subsection1.3.4 discussed configuration method that stores configuration files in the file system, but outside the application package.

3. Subsection 1.3.5 covered method that uses a configuration service to retrieve configuration values based on the environment where the application is deployed.

Different configuration methods have different advantages and disadvantages. When selecting specific solution for configuring the application with environment-specific configuration, then it should be discussed if reconfiguring the application at runtime is needed. Another important aspect to think about is how the application is deployed to different environments. When deployment process is partially or completely automated, then features of the deployment tools should also be kept in mind – it may be easier to integrate some environment-specific configuration approaches than others and, at the same time, to maintain or increase the level of automation related to the deployment process.

1.4 LiveRebel

1.4.1 Overview

LiveRebel is an application deployment tool that helps to easily deploy and update applications in different deployment environments, such as quality assurance or production. LiveRebel was initially focused on centrally managing Java application servers, but now it can also be used for deploying applications written in other programming languages, for example PHP, Ruby or Python.

LiveRebel consists of two main components: Command Center and agents. There is only one Command Center instance per each LiveRebel installation. Users always interact with Command Center instead of servers. Agents, on the other hand, run on each machine that needs to be managed by LiveRebel. While agents do the heavy-lifting, Command Center is responsible for coordinating the work of all the agents.

The Figure 1.1 provides a high-level overview of the LiveRebel architecture. Command Center application provides graphic user interface (GUI) for humans and REST API to applications for interacting with it. One Command Center can be used to manage many servers. On the Figure 1.1 there is only one Java application server and one "*File-based server*" (discussed in the next Subsection). Servers serve web applications over HTTP protocol and LiveRebel agents (also discussed in the next Subsection) in servers communicate with Command Center.

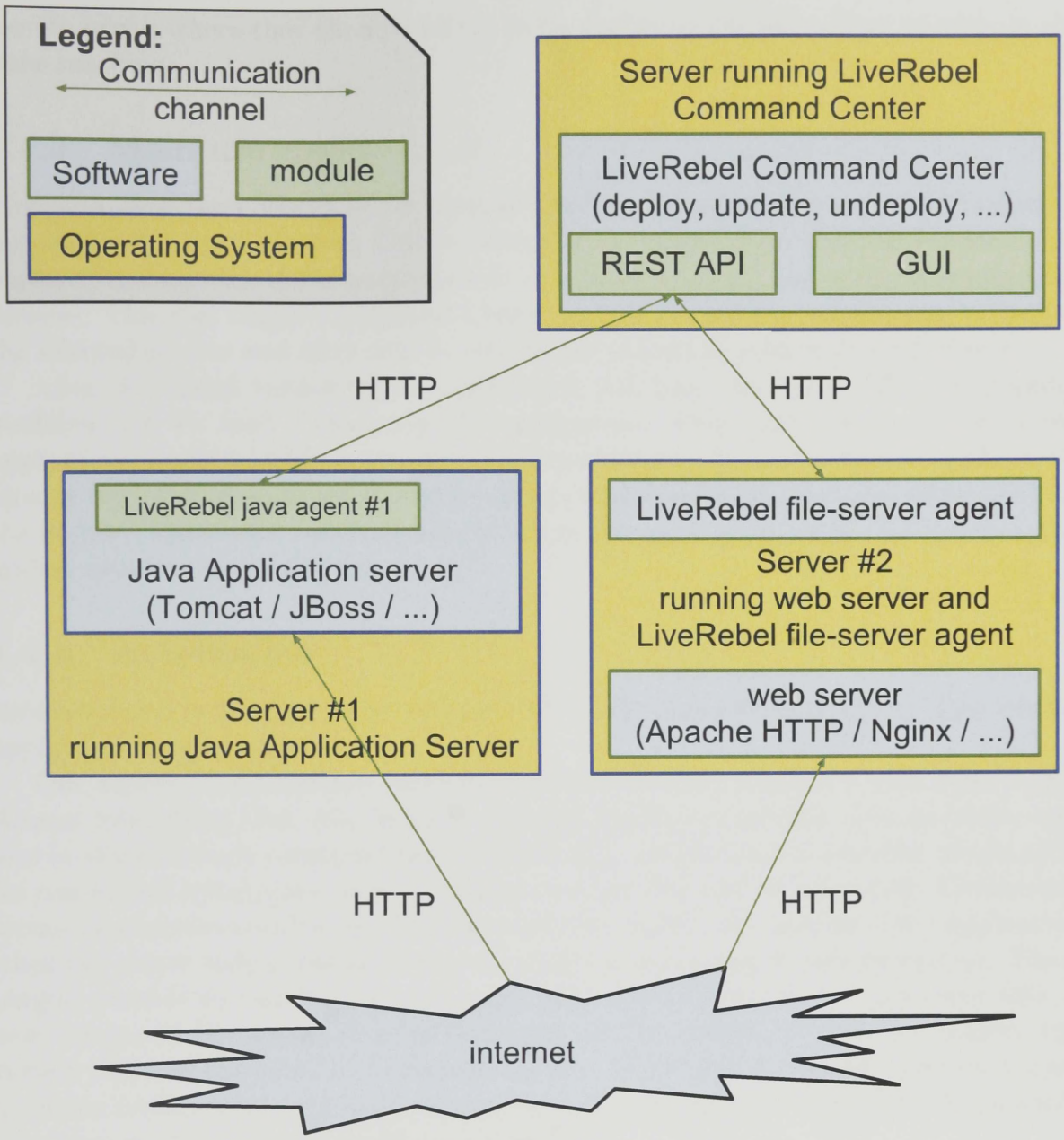

Figure 1.1: High level architecture of LiveRebel.

1.4.2 Integrations with servers

LiveRebel has deep integration with several Java application servers, such as Tomcat, JBoss, Jetty and Glassfish. Integration with application servers is implemented by using Java agent [13], which enables LiveRebel code to run in the same JVM as the server and enables to hook into the application server's internal features. LiveRebel agent is constantly listening for commands from Command Center. Command Center could, among many other things, let the agent to download new applications, deploy, update and undeploy them from the application server.

For other programming platforms LiveRebel provides a simple solution that involves standalone program, called *file-server agent* [14]. File server agent communicates with Command Center and can be controlled by Command Center like Java agents. It allows Command Center to upload application archives to managed servers, extract those archives and run scripts. These features provide means to ship files to the location in

remote server where they should end up to be usable by the web server running in the same machine.

1.4.3 Main use cases

First and most basic workflow for users of LiveRebel is uploading an initial version of the application to Command Center, selecting the application and the version to be deployed among with the servers where it should be deployed and starting deployment process. This will trigger Command Center to first upload the application to each of the selected servers and after that to deploy the uploaded version on each server.

After the initial version of the application has been deployed, the most popular workflow can be used – updating the application. Once again new version of the application should be uploaded to the Command Center. For updating the application, already deployed application should be chosen with the version that should be used for the update. Then after selecting servers where the application should be updated, the update process can be started.

1.4.4 Automation

LiveRebel also enables to easily automate the tasks that often need to be done during application version update.

One aspect of automation provided by LiveRebel is integration with other tools. Almost everything that can be done through LiveRebel graphic user interface, can also be done through command-line interface [15]. LiveRebel also provides plugins [16] for continuous integration servers, such as Jenkins [17] and Bamboo [18]. Continuous integration servers are often used to automatically build a new version of the application when the source code of the application is updated in the source code repository. These plugins provide an easy way to configure build and test process so that every time a new version of the applications is built and all the automatic tests are passed, the version could be uploaded to Command Center. LiveRebel plugins also provide means to choose servers where uploaded version could be automatically updated – for example in servers used for manual quality assurance.

Another example used very often is updating database schema in synchronization with the application code [19]. Developers can bundle files, containing changes to be done with the database, to the application archive – those files are automatically used by LiveRebel to update the database synchronously with updating the application.

The last example, is to copy some static resources to Content Delivery Network (CDN) [20] to be used by the application for static images, styles or scripts running in the browser.

These tasks could be described to LiveRebel as scripts inside application archive [21]. When LiveRebel finds scripts from the specific folder in application archive, then it will execute them at specified stages of the update or deployment process.

Scripts can also configure the same server where application is being deployed to, by copying, overwriting or even modifying files. There are several locations where scripts could take the files they use for copying, overwriting or modifying. The first approach would be to embed them in the application archive where configuration scripts could access them during deploying or updating. Another approach would be to place those

files to the artifact repository and let configuration scripts download them to the places where they are later needed by the application.

1.4.5 Configuration management feature

Based on the features described in the previous Subsection, the reader could figure out some solutions that could be used in several situations where configuration needs to be done before deploying or update. It may be true, but using deployment configuration scripts to configure the application with environment-specific configuration might not be the easiest tool. It would be more convenient if LiveRebel provided out of the box solution for configuring an application with the environment specific configuration.

Since LiveRebel manages and coordinates the deployment and update process it turned out that that LiveRebel could also provide means for configuring applications that would be relatively easy to use.

The contributions of this research are (*i*) a listing of features that would be helpful for environment-specific configuration management tool and (*ii*) implementation of environment-specific configuration management solution for applications managed by LiveRebel. Documentation about this feature with use cases are available on LiveRebel's documentation page [22]. The following Chapter introduces goals, architecture and implementation details of the environment-specific configuration management feature.

Chapter 2

LiveRebel's configuration management feature

Main contribution of this dissertation is implementation of environment-specific configuration management solution added to LiveRebel. There are too many goals to describe them thoroughly in the same section. For that reason the following Section just lists each goal with a brief description. Section 2.2 describes the architecture and implementation details of the configuration solution that also explains how the goals were achieved. The third Section of this Chapter explains decisions made for the implementation. The last Section mentions some products that have similarities with configuration management solution implemented for LiveRebel.

2.1 Goals for the feature

When designing application configuration solution, following goals were kept in mind (parentheses contain references to Subsections where those features are discussed in detail):

1. To provide the means to easily configure the application for each different environment (2.2.1, 2.2.9, 2.2.14).
2. To decouple environment-specific configuration values from the application archive (2.2.3, 2.2.9, 2.2.14).
3. To allow changing the application configuration, without requiring to build new application archive from scratch (2.2.2, 2.2.11):
    - The solution should provide means to use "configuration templates" - to allow to replace part of the file (for example placeholder inside textual configuration file).
4. To avoid disrupting development environment and process, e.g.
    - The development process should not get neither slower nor much more complex (2.2.5).
    - Using LiveRebel configuration management feature must be optional (2.2.11).

- The ability to set default values for everything being configured (2.2.5); The solution must be able to deploy the application to some environments without custom configuration package, but other environments could still benefit from custom configuration.

5. To avoid configuration duplication:
   - The solution should be able to assign the same configuration values at once to many environments (2.2.6, 2.2.9).
   - To allow using configuration values inside other configuration values (2.2.7).
6. To validate application archive and given configuration values before deploying or updating the application (2.2.8).
7. The ability to find out which configuration values are currently in use by the application that is deployed to a specific environment (2.2.13).
8. The ability to (re)configure application at runtime (2.2.14):
   - The application should be able to access the latest configuration values at runtime (2.2.14.1, 2.2.14.6).
   - The application should be able to receive notifications when the configuration has changed (2.2.14.1, 2.2.14.6).
9. The ability to secure sensitive configuration parameter values (2.2.12):
   - To allow encrypting configuration parameter values.
   - To allow decrypting configuration parameter values based on assigned permissions.

2.2 Architecture and implementation details

This Section explains how environment-specific configuration management feature was implemented for LiveRebel and how the goals set in the previous Section were achieved. There were more than one possible solution for several goals, but reasons for preferring one solution over the others are explained later in Subsection 2.3 – it is easier to discuss and reason about alternatives later, when the reader has a better overview of the whole environment-specific configuration management feature.

2.2.1 Central configuration management

Central configuration management approach was chosen to be able to easily manage the configuration for each environment. It means that environment-specific configuration is stored centrally in configuration management server. It made sense to embed configuration management server inside LiveRebel Command Center, because of several reasons. Firstly, Command Center is reachable from all the servers in all the deployment environments it is managing. Secondly, Command Center is coordinating the deployment and update process that could also contain activities related to configuration management.

2.2.2 Archive post-processor

To avoid rebuilding application archive from scratch with all the custom tools and dependencies configured for building that specific application, it was decided to decouple environment-specific configuration parameter values from application archive. A custom tool was developed to prepare the generic application archive for specific environment before deploying it.

From now on the archive processing tool will interchangeably be called either "archive post-processor" (as it post-processes the archive after it is built by build system), or "archive pre-processor" (as it pre-processes the archive before deploying it), just "archive processor" or "template engine"[1]. The process of adding the configuration to the application archive will be from now on called pre-processing or post-processing (depending on whether it is in the context of deploying archive or building initial application archive that does not have environment-specific configuration).

Archive processor needs to take two inputs: archive and environment-specific configuration and inject latter to the application archive produced as an output.

Archive processor was embedded into the LiveRebel agent running on deployment target servers. Both, application archive and target server specific environment configuration, are received from Command Center before deploying or updating the application on the server so that the archive processor knows exactly which configuration it should use before deploying or updating the application.

2.2.3 Identifying configurable parts in application archive

Processing tool has to identify parts of the application that it needs to modify with provided configuration. For that purpose placeholders were chosen to be used inside textual files to mark location where configuration parameter value should be inserted. Every placeholder consists of special prefix, configuration parameter name and suffix. These three components are used by the archive processing tool to identify the region that should be replaced by corresponding parameter value assigned in environment-specific configuration for the given environment. Example configuration "template" file that could be added to the archive that does not contain environment-specific configuration is shown in Listing 2.1. LiveRebel uses "`$LR{`" as a prefix and "`}`" as a suffix in placeholders, so for example "`$LR{db.host}`" would be replaced by value of "`db.host`" configuration parameter that is assigned for the environment whose configuration is used during post-processing.

Listing 2.1: Configuration file template with placeholders.

```
{
  "db.connection":
    "jdbc:mysql://$LR{db.host}/$LR{db.name}",
  "db.username": "$LR{db.user.name}",
  "db.password": "$LR{db.user.password}"
}
```

It is up to the application code what it does later with the post-processed file after the application is deployed. This example configuration file uses JSON syntax [23], but it could be as well any textual file, for example properties, XML, Javascript or

[1]Reason for calling archive processor as a template engine is explained in the next Subsection

even PHP source code file. Actually the archive processing tool does not even care if it is configuration file – even HTML page headings could be made configurable through managed configuration feature.

2.2.4 Providing values for configuration parameters

To define configuration for a specific environment, a simple properties file could be used where the property key defines configuration parameter name and property value provides the value to be used to replace placeholder with corresponding parameter name. In Listing 2.2 an example configuration is given that could be used to provide configuration parameter values for performance testing environment.

Listing 2.2: Example configuration file for performance testing environment.

```
db.host=qa.db.example.com
db.name=performanceQA
db.user.name=admin
db.user.password=secret123
```

By taking configuration template file from Listing 2.1 and environment-specific configuration values from the file shown in Listing 2.2, configuration template engine would produce a file, presented in Listing 2.3.

Listing 2.3: Example configuration file created by combining configuration template and environment-specific values.

```
{
  "db.connection":
    "jdbc:mysql://qa.db.example.com/performanceQA",
  "db.username": "admin",
  "db.password": "secret123"
}
```

Subsection 2.2.9 will discuss different ways that can be used to assign the environment-specific configuration values.

2.2.5 Accepting default configuration for non-intrusiveness

While using placeholders seems to be easy, the approach described so far has also a disadvantage. If the application is not pre-processed, but placeholders are used instead of usable configuration values, then the application would not receive valid configuration. It would be inconvenient for developers to pre-process the application every time before deploying to development environment, as it is done in environments managed by LiveRebel.

To minimise disruption in development environment, LiveRebel provides means to use default configuration files when the application is not pre-processed by LiveRebel before deploying. Default configuration files for the environments that do not use LiveRebel can be placed inside the application with the same paths that are finally used by the application. For development environment content of default configuration file (of template file that was shown in Listing 2.1) could be following, as seen in Listing 2.1.

Listing 2.4: Content of default configuration in file for template shown in Listing 2.1.

```
{
  "db.connection": "jdbc:mysql://localhost/demoAppDB",
  "db.username": "admin",
  "db.password": "secret"
}
```

Corresponding configuration template file should be placed under "`liverebel/expand`" folder[2]. Files from that folder are used during pre-processing to replace default configuration files only when application is deployed to the server managed by LiveRebel. After pre-processing liverebel folder is removed from the final application archive that will contain application code and environment-specific configuration.

Path of the template file should be constructed from archive processing tool's specific templates file folder and absolute path of the configuration defaults file. The template file corresponding to configuration defaults file "`app/conf/db.json`" would be "`liverebel/expand/app/conf/db.json`". The content of the corresponding template file could be the same as it was shown in Listing 2.1.

2.2.6 Avoiding configuration duplication

Usually all servers in the same deployment environment, for example production environment, also need to use mostly the same configuration parameter values. It may happen that for example a couple of configuration parameters may be set explicitly for each server, but most of the configuration values, such as database url, are the same for all servers of the same environment. For this purpose LiveRebel provides means to set configuration in different *configuration scopes*, as they are called in LiveRebel.

Server groups configuration scope could be used to assign common configuration for each server grouped together through Command Center. Server groups can contain any number of servers directly under that group and other server groups. Listing 2.1 shows an example of servers divided into two top-level server groups based on the application name. Both top-level server groups are divided into nested server groups based on the environments where those applications could be deployed. For both applications each deployment environment contains at least one server.

Having a server group and server configuration scopes provides means to assign configuration as granularly as needed[3]. In addition it provides means to set default value of configuration parameter in server group scope and allows to override them in more specific scope – in child group or server scope.

How configuration parameter values could be set for a server or a whole server group at once is discussed in Subsection 2.2.9.

2.2.7 Reusing configuration parameters inside other configuration parameter values

Another approach to avoid configuration duplication, is to reuse existing configuration parameters to compose new configuration parameters. For example, external service

[2]expand refers to the fact that placeholders, found from the files in that folder, will be "expanded" with corresponding configuration values

[3]Reasons why application level configuration assignment was not provided is discussed later in Subsection 2.3.1

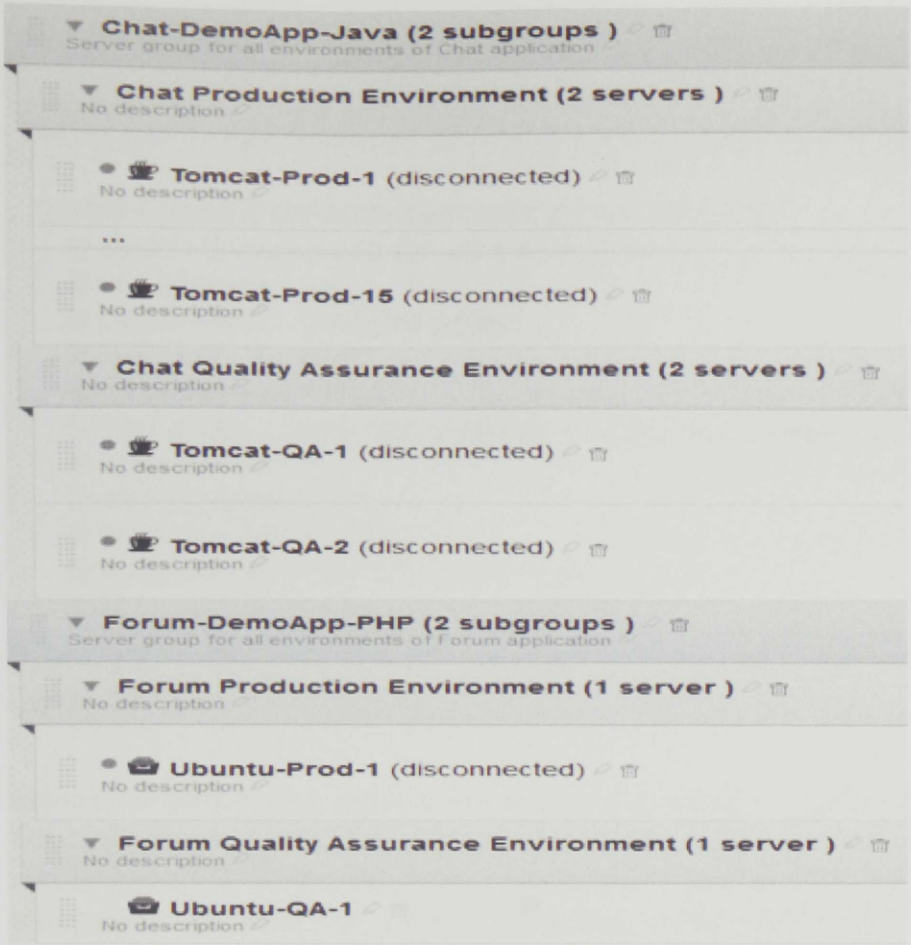

Figure 2.1: An example of server groups hierarchy for two applications (from LiveRebel servers list).

url could consist of several parts that may be needed in other places as well. Listing 2.5 demonstrates how the value of "`service.url`" configuration parameter could be derived from the values of other configuration parameters.

Listing 2.5: Example of configuration parameter value that reuses values from other configuration parameters.

```
service.scheme=https
service.host=external-service.example.com
service.port=8080
service.url=$LR{service.scheme}://$LR{service.host}:
    $LR{service.port}/
```

If an application contains a placeholder `$LR{service.url}` in configuration template file, then the placeholder would be replaced with the following string:

`https://external-service.example.com:8080/`

2.2.8 Validating presence of configuration before deploying and updating

In software development business it makes sense to validate as much as possible to avoid unpleasant surprises later when some functionality is used for the first time. The same

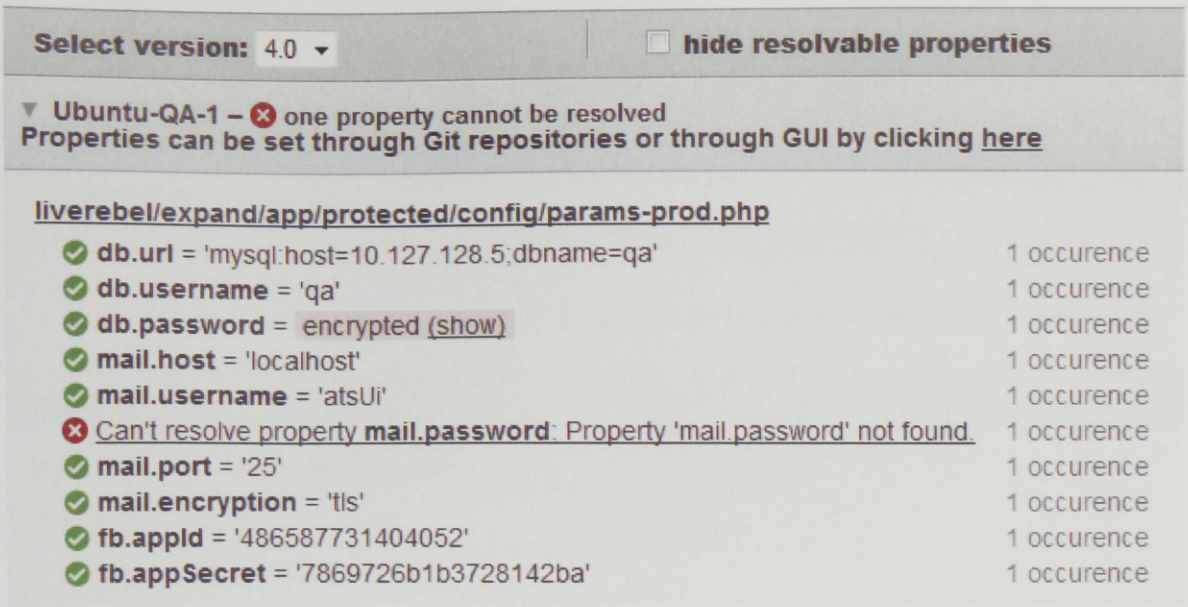

Figure 2.2: Verification before deploying/updating that all properties are resolved (from LiveRebel properties checking dialog).

applies to configuring application before deploying or updating it to a new version. For that reason LiveRebel analyses application archive when it is uploaded to Command Center. If during analysing configuration placeholders are found, then LiveRebel remembers the names of the corresponding configuration parameters found from the application archive. If the application archive did not contain any configuration parameters then the application could be deployed or updated without any configuration or further processing of the archive. However, if the application archive contains placeholders for managed configuration parameters, then for each server where application should be deployed or updated, the following check is performed. It is verified that the server has configuration parameter values for each placeholder found from the archive.

If there is at least one configuration parameter value missing for at least one server where the application will be deployed or updated, then deploying or updating could not be done on any selected servers and the user is informed about missing configuration parameter values as shown in the Figure 2.2.

On each row of the properties checking dialog that marks a missing property, there is a link to the dialog that allows setting properties (a screenshot of the dialog is also shown in the next Subsection – Figure 2.4).

2.2.9 Assigning configuration to environments

There are several ways to assign configuration parameter values to deployment target environments. One option is to clone Git repository [24] that is used to store configuration for a server group and servers [25]. The second option is to use GUI dialog that eventually also stores provided values in Git repository. Figure 2.3 shows a dialog that provides further instructions for both of those configuration options.

Figure 2.4 shows how property could be set through GUI to either all servers in the server group or to specific servers in that group.

The reader may have noticed "Property type" field that has not been discussed so far. In addition to properties that can be used during pre-processing of the application,

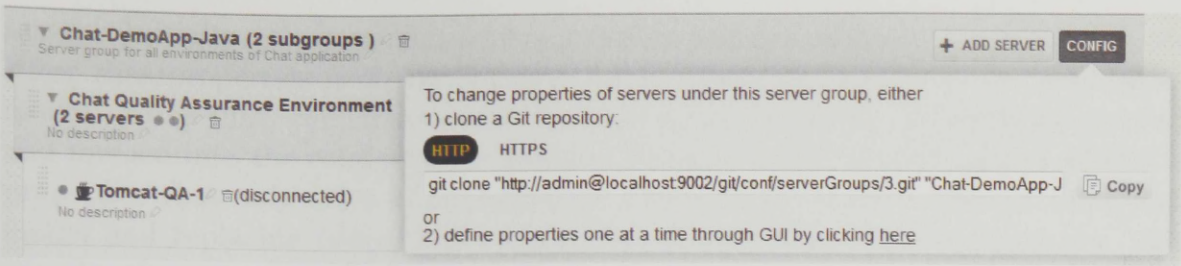

Figure 2.3: Configuration options shown in GUI (from LiveRebel servers list).

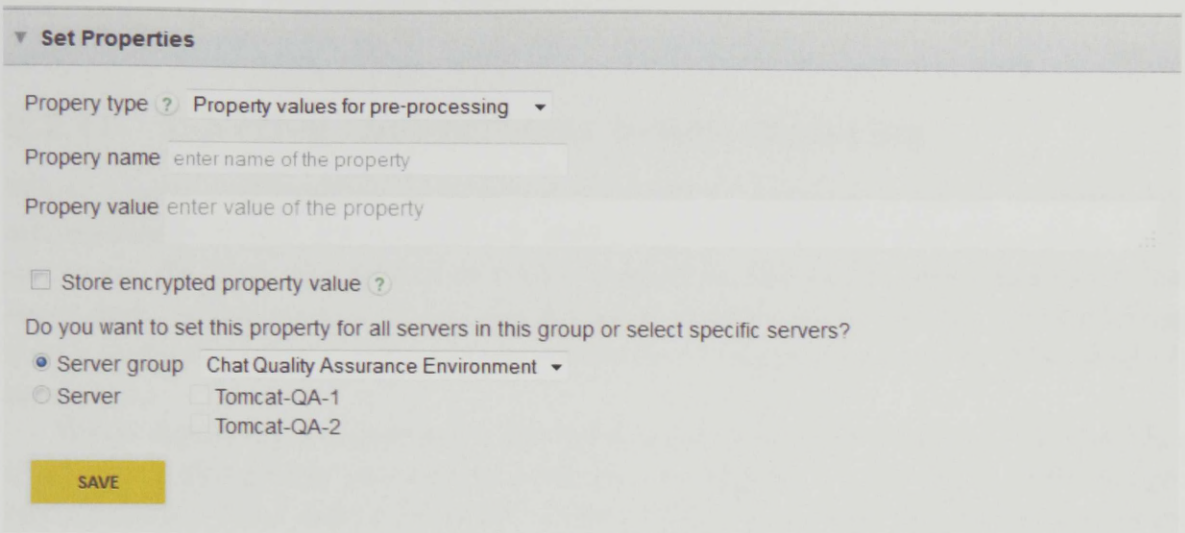

Figure 2.4: Dialog for setting configuration properties (from LiveRebel edit property dialog).

there are another set of properties that are available to the applications at runtime - that is discussed in Subsection 2.2.14. Encrypting properties will be covered later in Subsection 2.2.12.

As briefly mentioned before, internally LiveRebel stores environment-specific configuration in Git repositories that are also served through Command Center so the users could change files in repository. When the server group is created, then LiveRebel also creates new Git repository for that server group to be able to assign environment-specific configuration for all the servers under that server group. Each server group repository could also be used to assign configuration for servers directly under that server group. Configuration for each server is stored in its direct parent server group repository, under "`props/servers`" folder. Each server has an individual configuration file that is named after the name of the server. The configuration file that is used to replace the placeholders from the configuration template files are just named by server name and have "`properties`" file extension. The file name for assigning configuration for all of the servers under that server group is located in "`props`" folder of the repository and has "`group`" as a prefix instead of server name.

2.2.10 Detecting configuration for specific server

When Command Center needs to detect configuration for a specific server, then it finds out parent server groups of the server. Merging configuration for the server is started based on server group hierarchy from the farthest server group from the

server. Initially empty configuration model is created without any key-value pairs. After that properties from the farthest server group are added to the configuration model. After that, configuration of the next server group in the hierarchy is read in and merged into the configuration model. Merging in this context means adding new properties that only exist in configuration being added to the existing configuration model and replacing properties' values that already existed in configuration model. Finally after merging all the configurations of ancestor server groups, configuration assigned explicitly for the server is also merged into the configuration model the same way as described above. The result is complete configuration for the server that can be used by all applications to be deployed to the server.

2.2.11 Injecting configuration before deploying

When a new configuration is assigned to a server group or a server, LiveRebel will automatically detect for each server below the server group, if configuration for the server has changed. If configuration has changed for the server, then Command Center sends new configuration to the LiveRebel agent running on the server. LiveRebel agent stores configuration in the file system, where the configuration could be later used when needed.

When deploying or updating is initiated from Command Center, LiveRebel checks if all the configuration parameters used by the application are defined for the given environment. After that Command Center makes sure that the server has received the latest configuration assigned for the given environment, then it delivers the application archive to the server. If the application archive contains placeholders for configuration parameters, then the server uses configuration template engine to inject the configuration to the archive before deploying or updating the application.

2.2.12 Security considerations

Command Center has fine-grained permissions model, so each user could do or see only things that he or she is authorized. For example, one user could be able to only upload new versions of application to Command Center, but not deploy them to servers or update them – that is responsibility of another user.

Some configuration values may contain sensitive information – for example passwords. If configuration management solution would not provide ways to set values for sensitive configuration parameters, then the feature could be unusable for some organizations due to security policies they are following.

For this reason LiveRebel provides means to encrypt and decrypt configuration parameter values that should not be stored in configuration repositories as plain text. Encryption and decryption is implemented with Public-key cryptography [26], where private key for decrypting encrypted property values is available only in Command Center. Only users with sufficient permissions and LiveRebel deployment mechanism itself can decrypt encrypted properties values.

Encrypted property values are also stored in properties files of Git repositories, like regular unencrypted property values. However, to distinguish encrypted property values from unencrypted, a special prefix and suffix are added to the property value, so LiveRebel could recognize that it has to decrypt property value before using it instead of a placeholder. Encrypted property values, such as "`db.password`", are also hidden

in GUI by default as shown in Figure 2.2. Encrypted properties can only be decrypted by and shown to users with sufficient privileges.

2.2.13 Discover configuration changes for deployed applications

Configuration parameters could get new values after the application is deployed. However, that does not affect application configuration, as running application received configuration parameter values during pre-processing of the archive. As a result running application contains fixed configuration values instead of placeholders. It means that when the latest configuration stored in Command Center does not match configuration used by the application, it would be good to know it – maybe application should be updated with changed configuration.

For that purpose configuration change detection mechanism was implemented that shows a warning in Command Center GUI when deployed application contains at least one different configuration parameter value compared to the latest values that could be used by the application.

In addition, the configuration that is used to deploy the application could be downloaded as a zip-file that also contains additional information, for example when Command Center has sent that configuration to the server.

2.2.14 (Re)configuring the application at runtime

Sometimes the application may use features that could be needed to be reconfigured when the application is already running. Custom configuration solution could be implemented for each application, but LiveRebel Team Lead decided to provide also means for applications to access configuration parameter values that are managed through LiveRebel. Currently this feature has been integrated with Java applications, but in Subsection 2.2.14.6 it is discussed how it could be used for other programming languages as well.

2.2.14.1 Using runtime configuration Java API

Using this "runtime configuration" feature from Java applications is simple, as LiveRebelConfiguration class from lr-conf-api library[27] provides method "`getProperty`" that takes the name of the runtime property as an argument and returns String representation of the currently assigned value of the property. It also provides several convenient methods to get the property value and to convert it to some commonly used type, such as Boolean, Long and Integer. Example code in Listing 2.6 can be used by Java application to detect value of runtime configuration property with the name "`statistics.enabled`" and to convert it to boolean type.

Listing 2.6: Example how Java application could detect configuration value at runtime.

```
boolean enableStats = LiveRebelConfiguration
  .getPropertyAsBoolean("statistics.enabled");
```

To use this feature, lr-conf-api library must be found in classpath when compiling the application. When this application is deployed to Java application server managed by LiveRebel, then classes from lr-conf-api library are made available to all the applications and the library jar file is not required to be found in application libraries.

Also, when the application is deployed to a server that is managed by LiveRebel, then configuration values set in Command Center will be usable by the application.

When runtime configuration is changed for the server where the application was deployed, then the application could be even notified about the changes. Application could register configuration change listener for all configuration properties or for a single property as shown in Listings 2.7 and 2.8.

Listing 2.7: Example of registering for notification when runtime configuration has changed.

```
ConfigurationUpdateListener listener =
    new ConfigurationUpdateListener() {
  public void configurationUpdated(
      ConfigurationEvent event) {
    LrConfigSnapshot formerProps =
      event.getPreviousValues();
    LrConfigSnapshot newProps = event.getNewValues();
    Set<String> changedProps = event.getChangedPropNames();
    // got new configuration from LR-CC
    // "changedProps" contains names of properties
    // that were changed
  }
};
LiveRebelConfiguration
  .registerConfUpdateListener(listener);
```

Listing 2.8: Example of registering for notification when specific configuration parameter has changed.

```
ConfigurationUpdateListener singlePropertyListener
  = new SinglePropertyUpdateListener("prop1") {
  protected void propertyUpdated(String propName
      , String previousValue, String newValue) {
    // got new configuration from LR-CC,
    // "prop1" was updated
  }
};
LiveRebelConfiguration
  .registerConfUpdateListener(singlePropertyListener );
```

2.2.14.2 Detecting missing properties

As mentioned before in Subsection 2.2.8, it is desired to detect missing configuration before deploying or updating the application. For that reason each application version that uses runtime properties could declare all needed runtime properties in a metadata file, as shown in Listing 2.9.

Listing 2.9: Example of `WEB-INF/classes/declaredRuntimeProperties.txt` file that lists runtime properties that need to be defined for the environment before deploying or updating.

```
# Each line should contain only name of the property that
```

```
# must be defined for the application before deploying.
# Each property should be placed on separate line

# boolean - should application collect usage statistics
# (could affect performance)
statistics.enabled

# long (milliseconds) - how often new reports should
# be generated (generating reports takes long time)
report.autoUpdate.interval
```

Based on the information from this file LiveRebel can prevent deploying and updating application when any of the required runtime properties is not defined for the given environment. This has the same benefits as checking configuration parameters that are used for the placeholders inside textual configuration files.

In addition to listing used runtime properties names, this file could contain documentation about each configuration parameter type, how it should be used and what could be the consequences. All lines starting with "#" are ignored by LiveRebel, so they could contain text with instructions for people that need to set values for those properties before deploying the application.

2.2.14.3 Avoid forgetting to declare used properties

One flaw of the approach described so far is that a developer may forget to declare the property that was added during the development of the application. The solution that was implemented may not be perfect, but it works well with the assumption that using configuration parameter is tested. When application asks the value of a runtime property that is not declared to be used for the given version of the application, then an exception is thrown and it contains information about property name and how to properly declare it. By the time a new feature is implemented and a fresh application version is tested in quality assurance environment, all undeclared runtime properties that are used by the application should get detected and declared in the application archive for LiveRebel. That should give operations team confidence that they can provide all runtime properties that are used by the tested application version. It also means that when none of the deployed application versions uses runtime property, then runtime property could be deleted from LiveRebel configuration management repository.

2.2.14.4 Provide means to use undeclared properties

While declaring used runtime properties provides confidence to operations team, there may be use cases where developer could not or does not want to declare some of the runtime properties that the application could use. For that purpose developer could use another method, as shown in Listing 2.10, that does not throw an exception when the property is not declared.

Listing 2.10: An example of detecting value of a runtime property that could be undefined for the given environment.

```
String enableStatsStr = LiveRebelConfiguration
  .getPropertyUndeclared("statistics.enabled");
```

```
final boolean enableStats;
if (enableStatsStr != null) {
  enableStats = Boolean.valueOf(enableStatsStr);
} else {
  // this property is not defined for current environment,
  // using false as default value
  enableStats = false;
}
```

While there are several convenient methods for converting property value from String to other types, such as Boolean or Integer, similar convenient methods are not provided for getting and converting undeclared property value. The reason is to discourage using runtime properties that are not declared unless declaring runtime properties is not possible or practical. At the same time it encourages developers to declare all the runtime properties that the current version of the application uses so that LiveRebel and operations team could find out which configuration parameters are needed to be declared.

2.2.14.5 Using runtime properties feature without LiveRebel

Since LiveRebel is not meant to be used as a deployment and update tool in development environment (JRebel suits much better for development process [28]), then extra effort was needed to provide a convenient way to use runtime properties feature in the environments that are not managed by LiveRebel. The solution was to use a factory class [29] that creates a different implementation of ConfigurationResolver based on how environment is set up. When application is managed by LiveRebel agent, then the factory creates ConfigurationResolver implementation provided by LiveRebel agent. When the application is not managed by LiveRebel, and developer has provided custom ConfigurationResolver implementation[4], then latter is used. As a last resort, when neither of previously mentioned implementations are available, dummy ConfigurationResolver implementation is used that does not read actual configuration values from anywhere.

This approach allows developer to easily define development time values for runtime properties from a properties file of local file system. If that does not suit, then application developer could write custom implementation of ConfigurationResolver that corresponds best to the needs.

One thing to note is that when the application is deployed to a server that is not managed by LiveRebel, then lr-conf-api library must be added to the application. But since it is not needed in servers that are managed by a LiveRebel agent, then developers may want to customize the build process so this library would be embedded to the application only in a development environment. When using Apache Maven for building the application archive, then different build profiles [30] could be used, where only development profile would include lr-conf-api library as a dependency. As a result when building archive without "`-Pdevelopment`" command-line option, then lr-conf-api.jar would not be included in the application being built.

[4] lr-conf-api searches for org.zeroturnaround.liverebel.NoLiveRebelConfigurationResolverImpl class from classpath when application is not running with LiveRebel

2.2.14.6 Using runtime properties feature with non-java platforms

Regardless of the programming platforms used by the applications deployed through LiveRebel, LiveRebel agent installed to target deployment environment will get notified when configuration changes for that server. As a result LiveRebel agent downloads the given server specific configuration into LiveRebel agent specific folder in that server.

For Java applications lr-conf-api can be used to easily read runtime properties values and get notified about changes in configuration as they happen. Java developers using this feature do not need to be aware of internal implementation details, for example that the properties are read in from a specific file and how the notification mechanism is internally implemented – unless they want to provide custom implementation for the development environment.

In the future LiveRebel might provide similar tools for other programming languages as well, but until then those tools could be easily implemented by developers themselves if they need them. One thing that those developers need to know, is location of the properties file that gets updated by a LiveRebel agent when runtime configuration is changed in Command Center. If the file location is known, then reading in properties is already a simple task and listeners could be implemented by monitoring modification of the timestamp of the file.

The location of the runtime properties file could be detected by the application through placeholder `$LR{liverebel.runtimePropsFile}` that is replaced by the template engine like any other placeholders. However, "`liverebel.runtimePropsFile`" property is special in a way that it does not need to be set manually like property values for other placeholders. LiveRebel does not prevent deploying application if this special property is not set, because it knows that LiveRebel template engine can figure out the location of runtime properties file with the aid of other means.

2.2.15 Provisioning server for the application

While archive processing tool can be used to modify configuration files inside the application archive, sometimes the server where the application is being deployed, could also need provisioning. Some of the files changed during provisioning could also be used to configure the application.

For this purpose LiveRebel already provided means to execute scripts in the server where the application is being deployed or updated. As mentioned before in Subsection 1.4.4, these deployment configuration scripts can be embedded in the application archive and LiveRebel will execute them when needed. Scripts could be used to execute simple tasks, such as deleting a temporary directory. Scripts could also start other scripts or programs, for example Chef or Puppet mentioned before, to provision the server by installing and configuring other software.

Combination of configuration management feature and deployment configuration scripts provides comprehensive means to configure application for the environment where it is being deployed or updated.

Summary of the Section

One part of the environment-specific configuration feature developed for LiveRebel is the configuration server integrated to the LiveRebel Command Center. Firstly, it provides several ways, mentioned in Subsection 2.2.9, to manage environment-specific

configuration for applications that can be deployed to any deployment target environment that are managed by LiveRebel. Secondly, it also provides several ways to detect configuration assigned for the servers. Command Center prevents deploying applications to servers, where some configuration parameters used by the application are missing. This avoids program failures related to missing configuration values.

Other parts of the environment-specific configuration feature are "*mediators of the configuration*" from Command Center to the application. Both configuration mediators rely on LiveRebel agent running on the server where applications are being deployed. When configuration changes for that environment, then LiveRebel agent gets notified by Command Center, and the new configuration is downloaded to the server.

One configuration mediator is LiveRebel template engine that automatically injects the correct environment-specific configuration to the application archive right before deploying it as discussed in Subsection 2.2.11.

The second configuration mediator is a library that can be used by application to programmatically detect current *runtime* configuration values assigned for application running on that specific server. Currently the library is implemented for Java applications that allows to detect the latest value of the configuration parameter by calling a single method with the name of the parameter, as shown in Listing 2.6. Monitoring for changes in configuration parameter values is made convenient through registering configuration change listeners, as shown in Listings 2.7 and 2.8. A similar solution can be implemented for other programming languages as well, as discussed in Subsubsection 2.2.14.6.

2.3 Reasons for chosen architecture

There were many solutions, how several goals set in Section 2.1 could have been achieved. This section will discuss benefits of the approaches chosen for LiveRebel and disadvantages of alternative solutions that were not chosen for implementing environment-specific configuration management feature for LiveRebel.

2.3.1 No Application and global scopes

As mentioned in Subsection 2.2.6 current implementation provides two configuration scopes that can be used to assign configuration values - server group scope and server scope. Initial implementation contained two more configuration scopes: Global scope that could be used for all servers and in addition one scope for individual applications. Both of them were eventually excluded to keep things simple for the users of LiveRebel.

Global scope becomes obsolete when servers are grouped based on deployment environment – if needed, additional top level server group could be created for several deployment environments, to assign the default configuration.

Applications scope was left out because if a LiveRebel user wanted to deploy two applications to one server that would use for example different database connection urls set through managed configuration, then corresponding configuration parameter names could be prefixed for instance with application name.

2.3.2 Pre-processing in server, not in Command Center

Injecting configuration to the archive by the server instead of Command Center has the following benefits. Firstly, this way Command Center does not need to do pre-processing itself, which is important when many servers should start using the same archive. Each of those servers might have slightly different configuration and that way Command Center might slow down the deployment process if lots of servers are involved. On the other hand, when the archive is processed on the server, then all the servers that should start using the archive could inject configuration in parallel – that does not affect other servers. The second benefit is that when the configuration is changed and it is desired to just use the new configuration with the same application, then servers could reuse the previously downloaded archive without configuration to create a new archive with the latest configuration. This way network traffic and time spent to update the application are both reduced.

2.3.3 Configuration template files location

There are two conceptually different approaches regarding where configuration templates are placed. One approach is to include configuration templates in application archive and the other approach is to put them into a separate package. Advantages of both of the approaches depend on what needs to be configured.

Separate configuration packages could be very useful for provisioning the server with software related to the application being deployed. For example, an organization could use Apache HTTP Server for many web applications. If http server configurations for different applications should be similar, then one configuration of Apache HTTP Server could be described in "httpd.conf" file that uses placeholders instead of actual values so it could be customized for specific application. That configuration template could be used to configure the server by different applications before deploying the application for the first time. Also configuring MySQL database server could be done in the same manner. However, there are already excellent tools for this purpose, such as Chef or Puppet as discussed in Subsection 2.2.15.

On the other hand, when talking about configuring the application itself for different environments, then the configuration is highly specific to the version of the application. A new version of the application could introduce new configuration parameters or remove previously used configuration parameters. Also configuration file structure may have changed a lot which means that even if the same configuration parameters are used as in the previous version, the new version could not use configuration file created from an old configuration file template. For that reason it makes sense to search for configuration template files of the application from the application. When a new version is built, then the application archive would already contain correct configuration templates for that version of the application. Updating application configuration template files would become part of the development process, not the deployment process. Operations team does not need to do more than to make sure that all the configuration parameters used in the application have correct values. This approach eliminates two manual steps in deployment process. The first step would require creating or updating templates outside the application archive and the second step would be to make sure that the correct template is chosen for the correct application version being deployed or updated.

The fact that LiveRebel's environment-specific configuration feature does not pro-

vide out of the box features to configure related software running on the same server is not a problem in reality. There are other excellent tools that can be used for installing and configuring related software, as discussed in Subsection 2.2.15. Provisioning a server can even be automated during deployment process by combining deployment configuration scripts with provisioning tools. For example, before starting to deploy application, deployment script could start Chef with recipes that installs and configures Apache HTTP Server to the machine that is used for deploying the application and MySQL server to another server that could be used by several servers running the same application.

2.3.4 Implementation of lr-conf-api to access runtime properties

There are two different approaches for implementing a feature that allows application to access the latest properties values set through Command Center.

Escape configuration server uses the solution that relies on client-server model, where the application that wants to access property values is the client and server is Escape server. Every time when the application wants to know the latest value of a property or properties, it must contact Escape server that sends back all the property values set for that environment. This could become a performance issue, when the application needs to get notification about changes in configuration. The application could constantly poll for properties from the Escape server and compare the latest values to the values known from the previous response. However, if there are many applications or servers polling for changes at the same time from the same configuration server, then the configuration server itself could run out of resources to serve the configuration requests. Also the application that uses Escape configuration server needs to know the url of Escape server, the name of the environment where that application is currently running and the name of the application. These configuration parameters have to be set for the application through other means.

The approach used by LiveRebel was to rely on LiveRebel agent to fetch the latest configuration for that server whenever the configuration changes for that specific server. This is also a client-server model, where the server is Command Center acting as a configuration server. The client of the client-server model in this case is the LiveRebel agent running on the server managed by LiveRebel.

The LiveRebel agent does not constantly ask configuration parameter values from Command Center. When configuration has changed for that specific server, then Command Center will notify the agent that configuration was updated and it should get the latest values. That way neither the application nor the server has to periodically poll for configuration values. The second benefit is that Command Center, acting as a configuration server, does not have to determine configuration values for the server each time the server would otherwise connect to detect the latest values.

The third benefit of the approach used by LiveRebel is that the configuration is not transferred from the configuration server to the application (or a server running the application in case of LiveRebel's solution) unless configuration has changed for that configuration client (server managed by LiveRebel in this case). This way even when the application asks for configuration parameter value for the first time, then a remote call to Command Center is not done – runtime configuration is already stored in the same server and in case of Java applications using lr-conf-api library, those values are

even cached in memory for faster access.

This means that there is another benefit of using this approach – applications using runtime configuration never have to spend time on communicating with LiveRebel acting as configuration server. Also accessing configuration parameter value is done in constant time after initially parsing of configuration file, because values will be stored in cache for usage until updated configuration is received. When a new configuration is received and the application has registered configuration update listener, then the application will be notified right after the LiveRebel agent has received the latest configuration from Command Center. This approach enables configuration server to serve more applications monitoring for configuration changes.

2.3.5 Using placeholders

Some configuration frameworks, like Spring Framework, provide means to *"override"* default configuration parameter values defined for example in XML configuration files [31]. Explaining exactly how to use Spring's override configuration is not a subject of discussion in this study, but main concept is following. To use Spring's feature to override configuration, configuration object has to be accessed through Spring Framework's classes. Those classes parse the default configuration file and override definitions file and return configuration objects that are created by replacing default values with values overridden in override definitions file.

This approach can be useful in some situations, but LiveRebel does not provide similar out of the box solution for it. However, it does not mean that it cannot be used with LiveRebel. Usually using LiveRebel configuration template files with environment-specific property value placeholders, provides nearly the same end result in combination with default configuration files. Also existing third party configuration overriding solutions could be used in combination with LiveRebel environment-specific configuration solution. Custom configuration override solution can be used to describe default configuration file and override definitions for the configuration file. LiveRebel specific placeholders could be used instead of actual values in override definition files. Before deploying those placeholders are replaced by LiveRebel with environment-specific values. Configuration framework, used to apply override definitions and to read configuration, would not know that initially there were LiveRebel specific placeholders instead of actual values that were used to override the configuration.

Using third party solutions avoids creating new tools and relies on existing solutions that are already adopted by the developer communities. Combining third party configuration solution with LiveRebel environment-specific configuration feature also provides some advantages compared to using only third party configuration solution. One benefit is preventing deploying application without environment-specific configuration, as discussed in Subsection 2.2.8. Another advantage is a way to define configuration for all environments from one place.

2.4 Testing

Configuration management features that were developed for LiveRebel were covered with automated tests to verify that added functionality could be safely used in production environments. Written tests include unit tests and integration tests. In total there are more than 250 individual tests that try to cover all aspects of configuration

management feature. In addition to tests specific to LiveRebel configuration management feature, there are numerous tests for related features that are also involved when using LiveRebel configuration management features.

Implementing tests for GUI is planned, but so far not yet implemented. Configuration management feature is already released and does not have currently any known issues in the latest released version of LiveRebel.

2.5 Similarity with other products

While studying for present thesis, the author found some products that provide subset of environment-specific configuration management features implemented in LiveRebel.

Escape configuration server, mentioned in Subsection 1.3.5, provides only means to configure application at runtime. Similar concept with totally different architecture was implemented for LiveRebel as well. Disadvantages of the approach used by Escape configuration server were discussed in Subsection 1.3.5. Advantages of the approach taken by LiveRebel were mentioned in Subsection 2.2.14 and 2.3.4.

Another product worth mentioning is uDeploy [32]. UDeploy, like LiveRebel, is application deployment tool that mainly focuses on deploying and updating applications, but both of them also provide means to configure applications based on the environment where they are deployed. UDeploy does not provide means for application to configure itself at runtime, but otherwise there are many similarities with LiveRebel. Both products allow defining environment-specific configuration values for all managed environments in one place. In addition both products use configuration template files with configuration parameter value placeholders to create environment-specific configuration files. However, there are also many differences in the described feature. While LiveRebel stores application configuration templates in the application archive, for uDeploy configuration template files need to be bound to the selected application version and deployment target environment before deploying. Reasons why another approach was used in LiveRebel were discussed in Subsection 2.3.3.

Chapter 3

Future Work

As discussed in Chapter 2, the environment-specific configuration feature developed for LiveRebel can be divided into two sub-features for providing configuration parameter values for the application. One of them is using configuration template files containing configuration parameter value placeholders and the other approach is to programmatically ask the current configuration parameter values when application is running. At the time of printing the thesis, first approach is documented on LiveRebel user manual and available in the latest version of LiveRebel. The second approach is currently implemented, but not documented and it will be available in the next version of LiveRebel - that is LiveRebel 2.8.

In addition to adding the missing documentation for the features already implemented, there are some complementary features that could be added to the LiveRebel's environment-specific configuration management feature.

Another substantial feature that could be added is the ability to completely replace one configuration file with another file that could be assigned as part of the configuration to a specific application running on a specific server or a group of servers.

Also the graphic user interface could be improved. Some places could use the help of professional designer to make user interface visually more appealing. One visual component, used to inform the user about differences between present configuration values and configuration used during deploying, could provide more information. Currently it only shows a warning that configuration has changed, but it could also show exact configuration parameter names with values used by the application and the latest values assigned for that application. Currently this information can be gathered from Command Center, but it could be presented in a better way to provide convenient overview.

Conclusion

The purpose of this study was to investigate solutions to decouple environment-specific configuration from the application archive, to provide more convenient ways to change environment-specific configuration for the application. As a result of the theoretical part of this study, many goals were set for the environment-specific configuration management feature. All these goals have been achieved (*i*) by providing means to assign configuration for each deployment environment from one place and (*ii*) by providing ways to deliver those configuration values to the application. Two different approaches were implemented for applications to receive the configuration values assigned for the environment where they are deployed. One approach is used automatically by LiveRebel to replace configuration parameter value placeholders from the application archive before deploying the application. Another approach can be used programmatically by the applications that are already deployed to the servers managed by LiveRebel. When those applications are running, they can programmatically detect current configuration parameter values assigned to that specific environment via Command Center.

Environment-specific configuration management feature added to the LiveRebel application deployment tool aims to provide flexible means for operations teams to easily configure the applications they manage and to simplify the technical solutions used by developers to configure the same application for different deployment environments.

Keskkonnapõhise konfiguratsiooni eraldamine rakenduse arhiivist

Magistritöö (30 EAP)

Ats Uiboupin

Kokkuvõte

Iga rakenduse paigaldamisel erinevatesse keskkondadesse (nt arenduskeskkond, kvaliteedikontrolli keskkond, toodangkeskkond) on vajalik rakenduse konfigureerimine, mis sõltub keskkonnast, kuhu rakendus paigaldatakse. Rakenduse tarne ja paigaldusega on sageli seotud mitu inimest ning rakenduse (ümber)konfigureerimine toimub sageli käsitsi, mis on aega- ja ressurssinõudev tegevus. Kui keskkonnapõhise konfigureerimisega seotud info on rakenduse osa, siis peale konfiguratsiooni muutmist sageli ehitatakse kogu rakendus uuesti. Tavaliselt rakenduse keskkonnaspetsiifilise konfiguratsiooni muutmine ei nõua kogu rakenduse uuesti ehitamist, vaid üksnes konfiguratsioonifailide sisu muutmist. Käesoleva töö eesmärgiks on välja töötada lahendus keskkonnaspetsiifilise konfiguratsiooni eraldamiseks rakendusest, mis võimaldab lihtsustada rakenduse konfigureerimisprotsessi. Käesolev töö on jaotatud kaheks suuremaks peatükiks.

Töö esimeses peatükis tutvustatakse erinevaid metoodikaid, mida on võimalik kasutada rakenduse konfigureerimiseks. Samuti antakse ülevaade, millised on võimalused neid meetodeid kasutades teha rakendusele kättesaadavaks keskkonnaspetsiifilised seadistused. Lisaks tutvustatakse rakenduste paigaldamiseks mõeldud toodet LiveRebel, millele antud lõputöö praktilise osana lisati lahendus, mis võimaldab keskkonnaspetsiifilisi seadistusi tsentraalselt määrata. Keskkonnale määratud konfiguratsiooni kasutatakse automaatselt paigaldatavale rakendusele konfiguratiooni kättesaadavaks tegemisel, milleks rakenduse jaoks on realiseeritud kaks erinevat võimalust.

Teises peatükis sõnastatakse detailsed eesmärgid rakendusele LiveRebel lisatava funktsionaalsuse jaoks ning selgitatakse, milliseid lahendusi iga seatud eesmärgi saavutamiseks kasutati ning miks alternatiivsed lahendused kõrvale jäeti. Suur osa väljatöötatud lahendustest on juba LiveRebel viimases versioonis olemas ning ülejäänud kirjeldatud võimalustest on valmis ning muutuvad kättesaadavaks järgmises LiveRebel versioonis - LiveRebel 2.8.

Bibliography

[1] Damon Armstrong, Pro ASP.NET 2.0 Website Programming, Apress. 2005 [Online] http://dx.doi.org/10.1007/978-1-4302-0104-5_1 [19.05.2013].

[2] LiveRebel (application deployment tool), ZeroTurnaround. [Online] http://liverebel.com [19.05.2013].

[3] Command prompt (Cmd. exe) command-line string limitation, Microsoft. 2007 [Online] http://support.microsoft.com/kb/830473 [19.05.2013].

[4] Environment variable, Wikipedia. [Online] http://en.wikipedia.org/w/index.php?title=Environment_variable&oldid=555072703 [19.05.2013].

[5] Java Naming and Directory Interface (JNDI), Oracle. [Online] http://www.oracle.com/technetwork/java/jndi/index.html [19.05.2013].

[6] Escape (configuration server), Thoughtworks. [Online] https://code.google.com/p/escservesconfig/ [19.05.2013].

[7] Michael Jakl, Representational State Transfer, 2005 [Online] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.97.7334&rep=rep1&type=pdf [19.05.2013].

[8] Java Management Extensions (JMX) Technology, Oracle. [Online] http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html [19.05.2013].

[9] Gustavo Noronha Silva, APT HOWTO. Debian. 2005 [Online] http://jamsb.austms.org.au/courses/CSC2408/semester2/resources/debian/apt-howto.en.pdf [19.05.2013].

[10] Chef, Opscode. [Online] http://www.opscode.com/chef/ [19.05.2013].

[11] Puppet, Puppet Labs. [Online] https://puppetlabs.com/ [19.05.2013].

[12] Escape (source code of Java client API, Client class), Thoughtworks. [Online] https://code.google.com/p/escservesconfig/ [19.05.2013].

[13] Introduction to Java Agents, 2012 [Online] http://www.javabeat.net/2012/06/introduction-to-java-agents [19.05.2013].

[14] LiveRebel documentation, File-based servers, ZeroTurnaround. [Online] http://manuals.zeroturnaround.com/liverebel/guide/servers/file-based-servers.html [19.05.2013].

[15] LiveRebel documentation: Command Line Interface, ZeroTurnaround [Online] http://manuals.zeroturnaround.com/liverebel/guide/tools/cli.html [19.05.2013].

[16] LiveRebel documentation: Integrations, ZeroTurnaround. [Online] http://manuals.zeroturnaround.com/liverebel/guide/integrations/index.html [19.05.2013].

[17] Jenkins (continuous integration server) [Online] http://jenkins-ci.org/ [19.05.2013].

[18] Bamboo (continuous integration server), Atlassian. [Online] http://www.atlassian.com/software/bamboo/overview [19.05.2013].

[19] LiveRebel documentation: Database Migrations, ZeroTurnaround [Online] http://manuals.zeroturnaround.com/liverebel/guide/database-migrations/index.html [19.05.2013].

[20] P. Membrey, D. Hows, E. Plugge. Practical Load Balancing (Content Delivery Networks), Apress. 2012 [Online] http://dx.doi.org/10.1007/978-1-4302-3681-8_5 [19.05.2013].

[21] LiveRebel documentation: Deployment Configuration Scripts, ZeroTurnaround. [Online] http://manuals.zeroturnaround.com/liverebel/guide/configuration-scripts/index.html [19.05.2013].

[22] LiveRebel documentation: Environment Configuration, ZeroTurnaround. [Online] http://manuals.zeroturnaround.com/liverebel/guide/environment-configuration/index.html [19.05.2013].

[23] D. Crockford. Javascript object notation - JSON. 2006 [Online] http://www.ietf.org/rfc/rfc4627.txt, [19.05.2013].

[24] Git [Online] http://git-scm.com/ [19.05.2013].

[25] LiveRebel documentation: Defining properties for servers, ZeroTurnaround. [Online] http://manuals.zeroturnaround.com/liverebel/guide/environment-configuration/properties.html [19.05.2013].

[26] Arto Salomaa, Public-key cryptography, Springer-Verlag New York Incorporated, 1996 [Online] http://books.google.ee/books?hl=en&lr=&id=fgF0WOB_P-4C&oi=fnd&pg=PA1&dq=Public-key+cryptography&ots=0kL7rhAPRQ&sig=c2Dob467w4ZNvBsAa9nQIwZrQs0&redir_esc= [19.05.2013].

[27] LiveRebel Configuration Api library: LiveRebel public artifact repository, ZeroTurnaround. [Online] http://repos.zeroturnaround.com/nexus/content/repositories/zt-public/com/zeroturnaround/liverebel/lr-conf-api/ [19.05.2013].

[28] JRebel, ZeroTurnaround. [Online] http://jrebel.com [19.05.2013].

[29] S.J. Metsker, W.C. Wake, Design Patterns in Java, Addison-Wesley Professional, 2006.

[30] Maven documentations: Build Profiles, Apache. [Online] http://maven.apache.org/guides/introduction/introduction-to-profiles.html [19.05.2013].

[31] Spring Framework Reference Documentation: The IoC container - PropertyOverrideConfigurer, Springsource. [Online] http://static.springsource.org/spring/docs/3.2.2.RELEASE/spring-framework-reference/html/beans.html#beans-factory-overrideconfigurer [19.05.2013].

[32] UDeploy (Application deployment automation tool), Urbancode [Online] http://www.urbancode.com/html/products/deploy/ [19.05.2013].

Disclamer

Acknowledgments

First of all I would like to express my sincere gratitude to the people who during brainstorming helped me to identify different approaches that could be used to configure applications, come up with solutions that would probably fit best to LiveRebel customers' needs. Neeme Praks, the Product Lead of LiveRebel and Jevgeni Kabanov, the founder of ZeroTurnaround were very helpful.

I would also like to thank Helle Hein for advising me during writing the thesis.

Prerequisites

For understanding present thesis, familiarity with enterprise applications development and delivery process is recommended. Some examples are specific to applications developed for Java runtime environment. It requires the reader to have a basic understanding of web applications development and familiarity of deployment procedures, such as compiling source code, packaging application, deploying, undeploying and redeploying the application, starting, stopping and restarting the application server.

Non-exclusive licence to reproduce thesis and make thesis public

I, Ats Uiboupin (date of birth: 14.04.1984),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

    1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

    1.2. make available to the public via the university's web environment, including via the DSpace digital archives, as of **01.06.2018** until expiry of the term of validity of the copyright,

Decoupling environment configuration from application archive,
supervised by Neeme Praks and Helle Hein.

2. I am aware of the fact that the author retains these rights.

3. This is to certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 20.05.2018