Deployment automation for web portal components with metadata

Juri Urbainczyk
iteratec GmbH
Theodor-Heuss-Strasse 55
D-63225 Langen, Germany

Abstract: *The infrastructure of a complex web portal consists of many different technical components. This paper is based on experience in an industrial web portal project of a German car manufacturing company. The project has to manage an increasing number of parallel portal environments. The paper shows, how the portal components can be extracted from one environment and deployed to other environments automatically. Further, it details how they can be archived, in order to reuse them later in the same or another environment. The paper details how the system deals with the complex dependencies between the components and how to separate the environment-specific information from the components. Furthermore, it describes the implementation of the concept.*

Keywords: web portal, deployment, metadata, automation, configuration management, scripting.

1. Introduction

A web portal [1] is a web site or service that offers a broad array of resources and services to customers or users. Usually the portal integrated many web applications that implement the respective services or manage the access to the resources. Typically the users of the portal don't have to authorize each time they use another one of the web applications, because the portal itself handles single sign-on and session security context. The portal employs a complex infrastructure to fulfill these requirements. The infrastructure itself consists of various subsystems, which often are web applications again. Usually, there are subsystems which handle the authentication, others are responsible for displaying personalized menus to the user (authorization) while others again handle content or manage access to common resources.

In this paper, these subsystems are referred to as portal *components*. In literature, there are many different definitions of the notion component [2,3]. In our view, portal components are independent subsystems, which fulfill separate tasks for the portal and which access each other and can be accessed by other applications using clearly defined interfaces.

Using that notion of component, it is possible to define, which components make up the infrastructure of the portal at a certain time. The components have certain dependencies, i.e. that they reference and use each other in a way which can also be documented. But this infrastructure changes over time, when new versions of components are installed and old ones are deleted. Some components are removed completely or are replaced by others. The dependencies between the components change over time as well. E.g. the dependency to a database may only occur in the newest version of a component.

The number and different nature of the components, their dependencies and above all the fact that all of this constantly changes, leads to the large complexity of the portal infrastructure. This paper is about how to manage this complexity.

Chapter 2 introduces the business background for this paper and describes the portal and its subsystems. Chapter 3 clarifies the notions environment and market. Then, in chapter 4 the difficulties which arise from these circumstances are described. Chapter 5 states the objectives which were essential for this work. Chapter 6 explains how these objectives shall be achieved on a technical level. The following chapter 7 details the processes of archiving and deployment, as they are performed by the automating software. In chapter 8 we present the results and in the last chapter, we give an outlook on future work.

2. The project

The experiences for this report were gained in a web portal project of a German car manufacturing company. The portal, which was build up during that project, is one of many different web portals with identical infrastructure, i.e. they employ more or less the same components. The project developed the conceptional and technical concepts, employing only standard J2EE [4] architecture and JSP technology [5] for the portal (s. picture 1). Our group build up the infrastructure of the portal by configuring the purchased software or steering the internal software projects. Moreover,

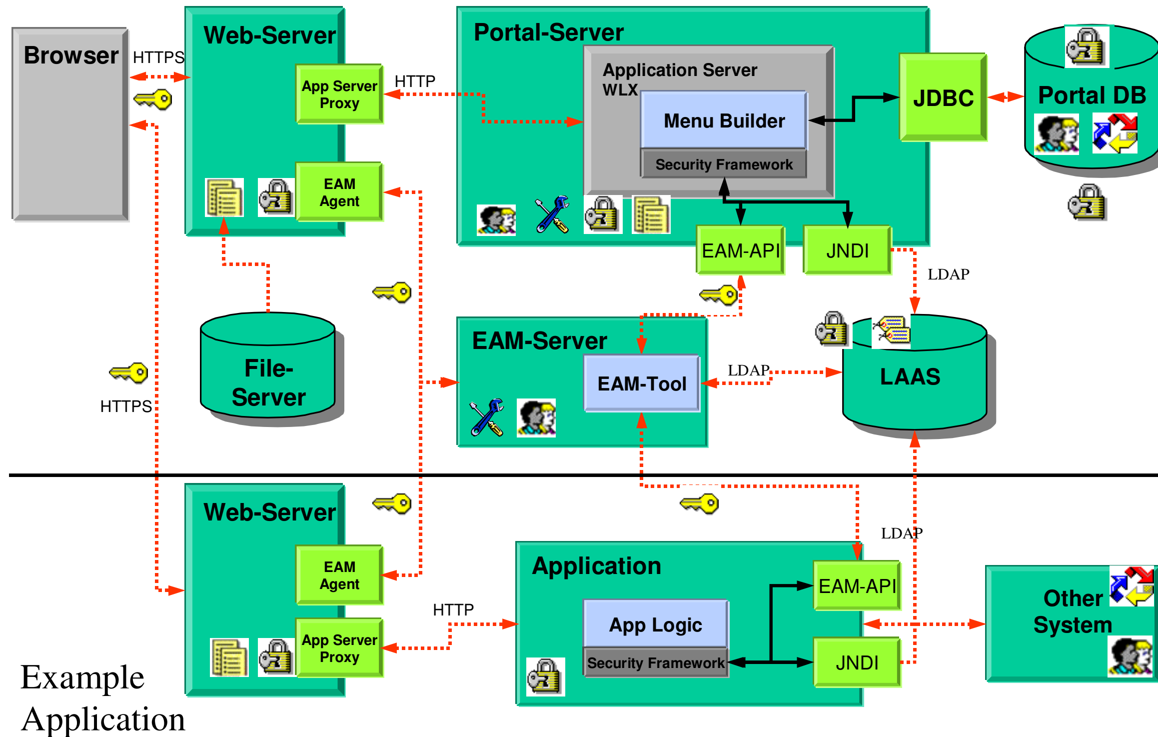
during development we were responsible for maintaining and extending the portal.

The portal integrates applications and content for a specific group of users, which is working worldwide. Applications are only loosely coupled with the portal: the application can access resources of the portal but not the other way round. The portal only references the applications by calling a URL.

Additionally, the portal offers infrastructure services to the applications, which are integrated into the portal. These services are:

- Single sign-on, authorization, authentication and creation of a common security context.
- Access to user data like roles and permissions.
- Management of user and permission data.
- Administration of the portal's infrastructure.
- Further services like bookmarking and sitemap. In this case "bookmarking" refers to storing the users favorite links in the portal's database (not the browser's).

To realize these services the portal utilizes various subsystems: user and permission data is stored in a LAAS (LDAP Authentication and Authorization Service) database [6]. To administer the data, custom web applications are used, which access the LAAS on the intranet. Authentication is realized by an enterprise application management (EAM) tool, which was purchased from an external company. This tool installs a plug-in in the web server and intercepts all HTTP requests. Only requests, which are equipped with an authenticated security context, can pass. If no security context is found, a login dialog is displayed. During logon the user data is read from the LAAS database.



Picture 1: The architecture of the portal and an integrated application

Authorization is realized using two Java applications, which were developed internally. One of them is responsible for the construction of the portal's menu, the other is used by other applications and other portal components to access common security data, like roles and permissions.

Every user is displayed an individual menu, which is derived from his permissions, which are again read from LAAS. Menus, which the user is not able to access, are not offered to him at all. The menu structure is stored in an Oracle database, which is accessed by the respective menu builder. With a Java Swing application, which was build in-house, the menu structure can be administered. There are further tools, which are needed to run the portal and its infrastructure, e.g. to migrate menus from one environment into another.

The portal features special HTML and JSP files, which are needed to customize the portal's GUI. Further files realize the log off from the portal.

The login is performed using functionality of the above EAM tool. Each portal has its own login pages, which again can be specific to certain envi-

ronments. The same is true for the change-password function.

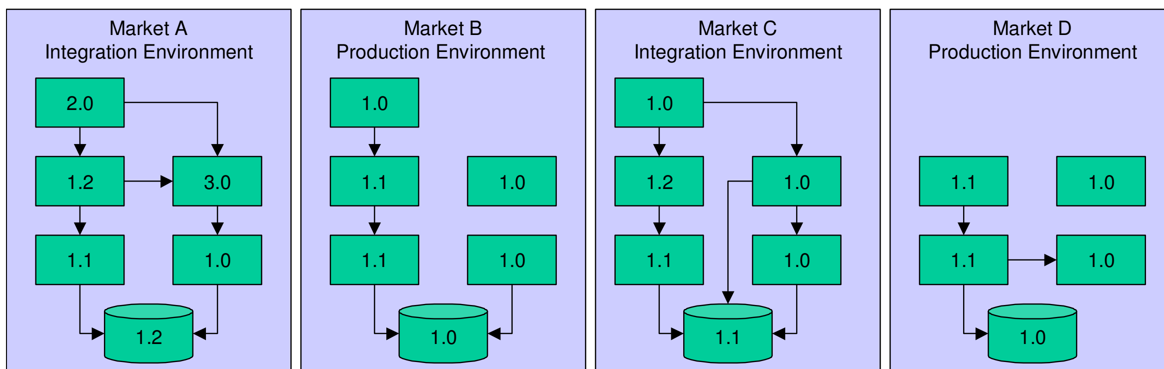
3. Environments and markets

The worldwide users of the portal are divided into many different sub-groups, which are organized by regional characteristics. These regions are called "markets". The markets usually use different applications or at least different versions of applications and therefore, each market needs a totally separate instance of the portal. There are at least one production and integration environment per market. Considering all markets the portal will finally end up with 80 or more environments.

Each of these environments is built from the same components. The components however can exist in different environments with different versions. E.g. in any environment there can be a different version of the EAM tool or the menu builder. Every environment could set up the components differently, as well.

Moreover, every market can show an individual menu structure, leading to a different content of the database. Integration and production environments differ as well, because the applications are integrated in the integration environment first and moved to production later on. Additionally, there is certain content, which only exists in production, like news and the bookmarks of the portal's users.

Therefore, every environment has a totally different infrastructure, consisting of an individual group of components, which is potentially set up differently (s. picture 2).



Picture 2: Environments having different components and component dependencies.

4. Difficulties

When our project started we began just with one market, containing only two environments. Whenever a new environment or, rarely, a new market had to be installed we did that manually. At first, we installed the web and application servers and the databases. Then we received the current versions of the components from the developers and deployed them. After that we adjusted the environment specific settings of the components by manually editing their files or database tables. All in all, this could take days, or weeks if we also count the time to wait for licenses.

When we deployed a new component to a certain environment, we had to manually check which version was installed. Moreover, we had even to look up the versions of *all other* components, because we had to guarantee that they would fit to the new one. Since it was not evident what component was dependent on what other, we often had to figure that out as well.

For what is worse, the components were not easily to be separated from another. Some components contained data, which was part of another component, like a hard coded name of a web application. I.e. that the deployment of one component could have profound consequences on others.

Furthermore, there were also many dependencies between the components and the environment in which they were deployed. Some components had to be set up especially for the environment using data like the cluster name, IP addresses, database connect strings, port numbers and so on.

This process was very error-prone and time-consuming. When the number of markets and environments increased, it became evident that this process was no longer feasible. The portal infrastructure just was too complex and the environments were too abundant to manage them manually.

To sum it up, we had the following problems:

- We didn't know which components were installed
- The dependencies were not documented

- There was no clear separation between components
- The components were polluted with environment data

These facts became problems when the number of environments crossed a certain limit, because we were no longer able to install and manage the infrastructure of the portal in time.

5. Objectives

It became clear, that we needed an automated management of portal components and environments. Every environment consists of a *configuration* of components, i.e. a defined set of components with defined dependencies. What needed above all was control over the configurations of our environments. That should help us to achieve the following aims:

- We want to install new environments within one day or faster.
- It must be possible to determine the state of an environment quickly in terms of which components are installed in what versions.
- To do that, it must be possible to identify every component with its name and its version. The version must be unique and related to a source code version of the component.
- We must be able to quickly build up a properly running environment with a defined state of components. That can also be a old state, which existed in a different environment some time ago. This is necessary e.g. in order to recreate errors which only occur with certain combinations of components.

6. Approach

At first we had to define what exactly our components were, because at that moment they were not at all clearly separated. The following conditions must hold for a portal component:

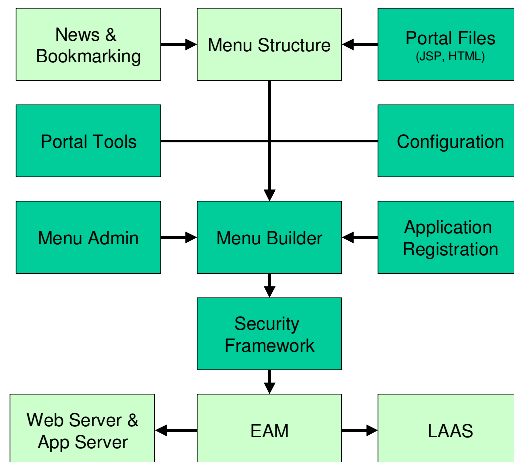
- The subsystem is developed further independently of the others – it forms its own versions.
- The subsystem may be installed in different versions in separate environments.

These conditions lead to the component hierarchy, which is shown in picture 3. Transitive dependencies are not displayed there.

The components in dark green color are included in our approach. These are the following:

- The *portal files*, which are used for login, logout and customization for different markets.
- The *portal tools*, which are needed e.g. to transport menu data between environments.
- The *menu builder* application, which constructs personalized menus.
- The *menu admin*, which is needed to administer the menu structure.
- The *application registration*, which is used to integrate new applications into the portal.
- The *configuration* component, which store configuration data for other components.
- The *security framework*, which is used to access the LAAS database and security functionality.

The menu structure and the news and bookmarking components are different, because they store runtime data and therefore need special consideration, which will certainly happen in a next step. Web and application servers were not considered in our concept. Due to the organization of the project those servers are not installed by our group. In our view, they belong to the runtime system of the portal. Every environment is based on potentially different settings of the portal's runtime system. The following parts belong to the



Picture 3: The component hierarchy

runtime system: the web and application server, the Oracle database and the LAAS database.

In order to achieve the above objectives, we decided on the following points:

- The environment data must be separated from the components. This is necessary in order to exchange components between environments.
- The dependencies between the components must be known and documented. This is necessary in order to automate there deployment.
- There must be a global repository for components. A component with a certain version can be extracted from the repository any time.

- There must be a mechanism to store an existing configuration (archiving) into the repository and to retrieve it from there at a later time (deployment).

The archiving and the deployment should be realized through the use of UNIX shell scripts. The global repository for the components should be a dedicated directory. It would be accessible for all relevant user accounts and from all machines, which are involved with the portal. The archives of the respective components should be stored there as tar files.

```
<clusterconfiguration>
<machine name="##CLTR_SVR1_NAME##">
<url>http://##CLR_SVR1##:##CLR_SVR1_PORT#
#</url>
</machine>
<machine name="##CLW_SVR2##">
<url>http://##CLR_SVR2_NAME##:##CLR_SVR2_
PORT##</url>
</machine>
</clusterconfiguration>
```

Listing 2: Tags replacing environment data

In order to get the components manageable, their structure had to be changed: every component has to describe itself and its configuration settings with metadata [7], i.e. information on itself. Thus, a component now consists of three parts: two files, which hold the metadata and the instance data of the component itself.

The first of the files is called *id file* and contains the following information: the name of the component, its version and a list of all objects, which belong to the component. The id file has XML syntax can be regarded as a kind of deployment descriptor [8] for the component.

The id file describes the location of all objects of the component in relation to the installation location of the component. This is helpful with archiving and restoring the component. For every object the attribute *type* is stored [9]. It describes, which type of object it is, e.g. files or database objects. This information is necessary to deduce which tools are needed to archive or restore a component. If the object is a database object, oracle export mechanisms are used. The version number stands for a concrete version of the component.

If a component is deployed to an environment its id file is stored at a dedicated place. There you can find all id files of all components installed in this environment. This is a precondition in order to save a configuration automatically.

The decision, to work with the id file, was made, when it became clear, that there was partial overlap in the file structure of the components. if one just archived the directory tree of a component, one would most certainly also include files of other components, which is not feasible.

There is also a second file with metadata, the so-called *config file*. It describes the location of all environment-dependent data in the objects of the component. This is needed for *normalization* of the component. A normalized component does not contain any environment-dependent data [10]. This is achieved by extracting all environment-dependent data from the objects of the component and then replacing it by tags (s. listing 1). The normalization is performed when the component is archived, i.e. an installed component is not normalized.

Nonetheless, during deployment the environment data is needed. Therefore, all information specific to one environment is gathered in one file, the *environment properties* file. One such file exists for every environment (s. listing 2). In this file you find the concrete IP addresses, port number, machine names etc. of the environment's runtime system. This step decouples the components from the underlying runtime system by removing the dependencies from the components and transferring them to the environment properties file.

```
##MARKET_LC## = XY
##MARKET_BS## = /dportal/##MARKET_LC##
##SMLOGINDOCROOT## = .
##ENV## = INT
##PERMISSION_SERVER ## = permserv2
##PERMISSION_PORT## = 7843
##MULTICASTIP## = 192.0.1.111
##ADMINSERVER_NAME## = pwpstep3
##ADMINSERVER_PORT## = 9115
##ADMINSERVER_PORTSSL## = 9223
##SERVER1_NAME## = pwpstep3
##SERVER1_LISTENADDR## = 140.23.23.1
##SERVER1_PORT## = 1123
##SERVER1_URL## =http://##SRVR1_NAME##
##SERVER2_NAME## = pwpstep4
##SERVER2_LISTENADDR## = 131.110.45.22
##SERVER2_PORT## = 9928
##SERVER2_URL## =http://##SRVR2_NAME##
##APPID## = dportal_##MARKET_UC##
##LDAP_SRVR## = idbldap.fra
```

Listing 1: Environment properties

We also discussed another option, which would have worked without config file: why not simply *replace* the old environment-dependent data with the correct and new data when deploying the component? This could be possible, but would have meant extensive scanning algorithms and techniques in order to detect the environment-dependent data. E.g. one could have assumed, that numbers of a certain structure, separated by points, are IP-addresses, which would have to be

replaced by the correct IP-address. But this seemed to be too much work and still too error-prone to be feasible.

After executing the above steps, the components are clearly separated from each other and from the environment data. Now, we could implement the automated archiving and deployment procedures.

7. Implementation

Archiving

The archiving (the process of extracting the components from the environment and storing them in the repository) is realized by several scripts. The script always works with respect to one environment, the so-called *source environment*. Either it archives all components, i.e. the whole environment, or it archives only one certain component. The control information is passed by arguments to the script when invoking it.

The script performs the archiving with the following steps:

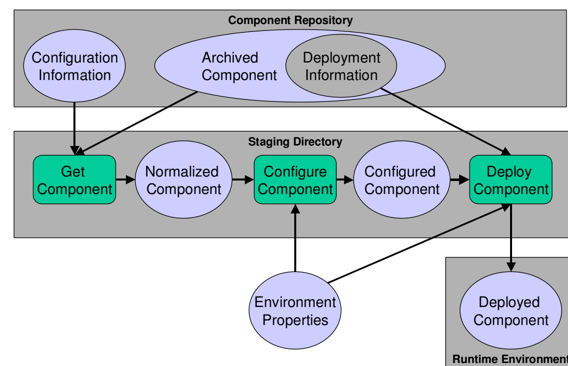
1. The directory with the metadata of the installed components is read. The script decides for every component if it has to be archived. If the component is already in the global repository with exactly the same version then it will be skipped.
2. The necessary components are copied to a temporary directory – the so-called *staging directory*. All further work is done there on the copy. Thus, the archiving does not influence the source environment at all.
3. The components are normalized one after the other. Now the information from the metadata is used to remove the environment dependent data from the objects. The environment specific data is replaced by tags.
4. Now the components are stored in the archive. Every object of the component is treated as suitable for its type, which is read from the metadata. Eventually, all files are packed with the UNIX tar command to one big archive. The tar file gets a name, which contains the name of the component and its version. The result is a complete *component archive*.
5. The tar file is stored in the global repository. Then all files generated in the process are deleted from the staging directory.
6. The information about the archived configuration (which components were installed with what version) has to be archived as well. To this end, another file is generated. It contains the following information: the versions and names of the component archives contained, name and path of the source environment, date and time of the archiving, the account name of the user who started the archiving and the machine name where it was executed. This file is stored in the repository together with the file containing the technical settings of the environment.

As result of the process, there is a set of one or more tar files in the repository. Every tar file represents one archived component. Additionally, there are two files containing information about the state of the source environment at the time of archiving.

Deployment

The deployment of components has to be started manually, since there must be a concrete reason for it. E.g. a totally new environment for a new market has to be installed.

Like the archiving, so is also the deployment realized by scripts. One *target environment* and one configuration must be passed as arguments. The components, which make up the given configuration, are read from the global repository. It does not matter, if the components got there by an earlier archiving or if they are newly developed and stored to the repository for first use.



Picture 4: The deployment process

If an already existing environment shall be overwritten with a new configuration, it has to be initialized before. Especially the application server has to be shut down, which can be done automatically with a separate script. During deployment of the new configuration the following steps are performed:

1. The script determines the components to be installed from the given configuration (s. picture 4).
2. The necessary component archives are looked up in the repository. If they are not found, the process is aborted here.

3. All needed component archives are copied from the repository to the staging directory. At this step, the components are still normalized.
4. In the staging directory the components get denormalized. This is achieved by looking up the tags generated at archiving time and replacing them with environment specific data. The script reads the environment specific data from then environment properties file, which is passed as argument.
5. Eventually, the denormalized components are deployed to the target environment. It's not necessary to follow a certain sequence when deploying.

After this procedure the environment already is fully functional. All components are exactly at that state with which they were archived to the repository. The application server can be started up and the portal can be used.

8. Results

Using this archiving and deploying procedure described above makes it possible to install a new environment just by „pressing a button". Thus, the amount of work necessary to build up a new market is reduced from days to just an hour or even less. The results are reproducible and can be repeated as often as required. Now we have the possibility to install many environments with the same configuration. Furthermore, we are able to react quickly to customer requirements. Changes to environments can be done in minutes. If it is necessary to install a different version of a component in order to remove an error, this can be done very quickly. All in all, the maintenance of the environments would not be possible without the deployment automation.

Furthermore, we now have a possibility to revive old configurations, which were formerly archived. This is very helpful when looking for errors or when trying to answer support requests.

9. Future work

Until now the dependencies between the components cannot be checked when deploying, because the necessary information is not available. It makes sense to define the dependencies in the metadata, e.g. in the id file. If this information would be accessible, the deployment script could assert that the given dependent components would be installed as well. If this condition could not be met for all components, the process could abort with an appropriate error.

As a next step we also have to think of archiving and managing of portal contents, like the menu structure and the news and special runtime data like the bookmarks. The menu structure poses one special problem: the numbering of the versions must be able to cope with variants (parallel versions).

The management of the already archived components must be enhanced as well. Until now the normalized components are stored as tar files in a global directory. In the long run the components shall be stored in a true CM system, like Continuus. The benefit would be, firstly, the automated version control. Secondly, the component could be linked directly to the related version of its source code.

10. References

[1]: Enterprise Portale JavaMagazin 12.2002

[2] Gruhn, Thiel: Komponentenmodelle DCOM, Javabeans, Enterprise Java Beans, CORBA Addison-Wesley - Pearson Education

[3] Cunningham Component Definition
http://c2.com/cgi/wiki?ComponentDefinition

[4] Java 2 Platform Enterprise Edition Technology
http://java.sun.com/j2ee/index.jsp

[5] Java Server Pages Technology
http://java.sun.com/products/jsp

[6] LDAP Definition
http://www.webopedia.com/TERM/L/LDAP.html

[7] Metadaten
Deklarative Programmierung durch erweiterbare Metadaten, Object Spektrum 02/2003

[8] Deployment Descriptor
Enterprise Java: Konfigurationsbeschreibung für EJB-Komponenten, Java Spektrum 06/1999

[9] Zhichen Xu, Magnus Karlsson, Chunqiang Tang, Towards a Semantic-Aware File Store HP Laboratories

[10] Donald D. Cowan, Carlos J. P. Lucena: Abstract Data Views: An Interface Specification Concept to Enhance Design for Reuse IEEE Transactions on Software Engineering archive Volume 2, 1995