Advanced Enterprise Portals

by Walter Kriha

Advanced Enterprise Portals

by Walter Kriha

Published 3 May 2001

Table of Contents

List of Figures

List of Tables

List of Examples

Dedication

Written against the end of a larger project, this paper contains many ideas from my friends with whom I've been building this enterprise portal. I would like to say thanks to our project team

and to our project leaders

Preface

This document describes the specific problems of a large scale high-volume portal site (in this case a portal for a large international bank). By showing what is necessary to achieve a large scale high speed portal - both in architecture as well as implementation - it questions whether this effort is justified for all portal needs. It is not an introduction to Enterprise Portals. It assumes working knowledge of portals and associated technology.

- You should know the Java programming language reasonably well.
- You should know XML and backend access technologies.
- You know a Web Application Server like tomcat or IBM Websphere. The physical architecture of Internet Applications (including DMZ) should be familiar - including load-balancing and clustering

If you're just getting started the resources chapter contains links to a lot of publicly available papers you might consider reading

Chapter 1. Introduction

Just about a year ago in May 2000, AEPortal (Advanced Enterprise Portal) started as a typical web project: Put a wrapper around some existing services, retrieve some existing data sources and hide everything behind a common GUI. Clearly the functional requirements were the most important ones at that time.

Based on a rudimentary framework (similiar to an early version of STRUTS from apache.org), modeled roughly after the J2EE architecture the team started to write handlers, models and JSPs.

It looked like AEPortal would become an Enterprise Portal.

In the middle of October last year – when load tests and preparations for deployment started – the non-functional requirements made a sudden and violent appearance and they continue to dominate AEPortal till this very day.

Much has changed since then. AEPortal has been extended in many ways to cope with the non-functional requirements – going far beyond a regular web application. This document describes the portal specific aspects of AEPortal with a special focus on reliability, stability, performance and throughput.

At the end organizational problems that had a major effect on AEPortal are described as well.

Purpose & Scope

As an enterprise portal AEPortal goes beyond a simple web application architecture. It relies on internal multithreading, read-ahead and does a massive amount of caching to concentrate a large number of external services onto a personalized homepage - while still being responsive.

Being a portal AEPortal needs to combine transactional features (e.g. Telebanking) with publishing features (e.g. News). Right now it does a fairly good job on the transactional side but it is very weak with respect to publishing. This needs to change quickly and the document tries to highlight the problems and give hints on possible solutions.

This document also explains the technology used in AEPortal. It discusses critical aspects e.g. in case of external service failures. These topics are not covered in the Infrastructure Architecture Document. (24)

Caching plays a major role in AEPortal. Every site that delivers dynamic AND personalized content has a severe performance problem currently solved or mediated through large hardware investments.

Example 1.1. A little hardware at Charles Schwab

A recently published paper on the portal architecture at Charles Schwab shows that IBM delivered 500! Multiprocessor Unix stations to run the personalized portal. Of course the fact that the portal still runs CGA technology certainly accounts for a larger number of workstations - still - the numbers are frightening both financially and from a system management point of view

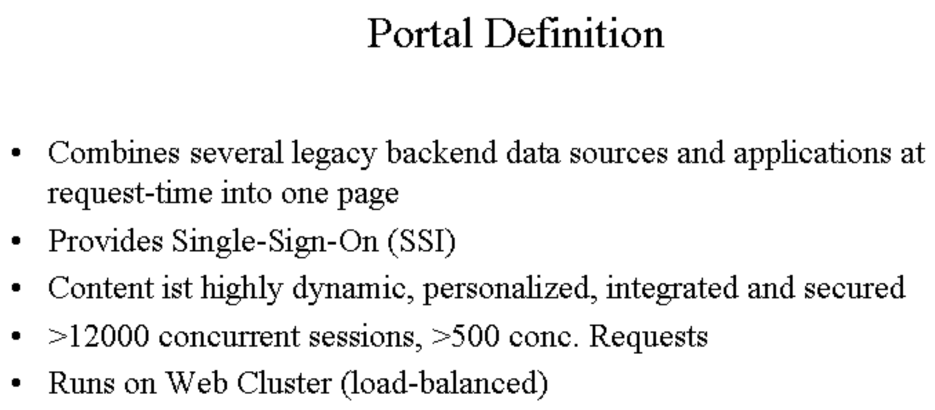
Caching of dynamic AND personalized content is both hard to do and an absolute must for such sites. The overall architecture of a high-volume portal is more or less dominated by caching issues. This document collects what AEPortal already does with respect to caching as well as what still needs to be done.

Another goal of this document is to collect ideas from within the AEPortal team or our friends. So please do not hesitate and add to it. There are still a lot of features missing that are vital for an enterprise scale portal (meta-information handling, search features, service development kit, maintainability in production etc.)

Chapter 2. What is an Advanced Enterprise Portal?

Definition

Figure 2.1.



Portal Definition

- Combines several legacy backend data sources and applications at request-time into one page
- Provides Single-Sign-On (SSI)
- Content ist highly dynamic, personalized, integrated and secured
- >12000 concurrent sessions, >500 conc. Requests
- Runs on Web Cluster (load-balanced)

Example

Figure 2.2.

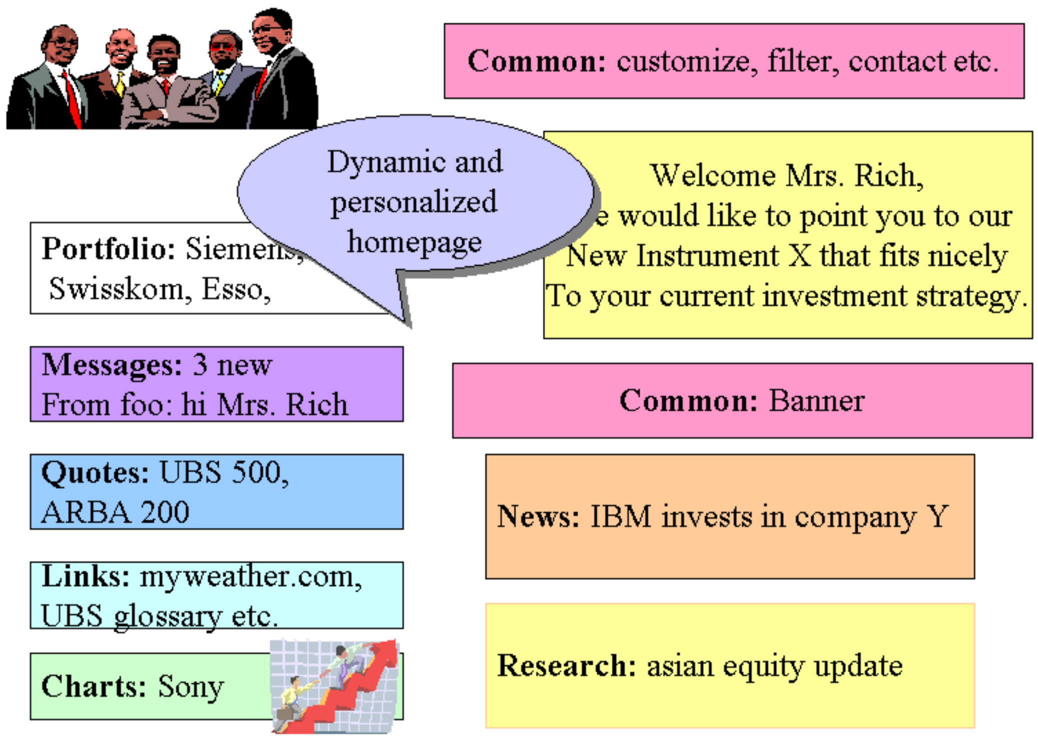
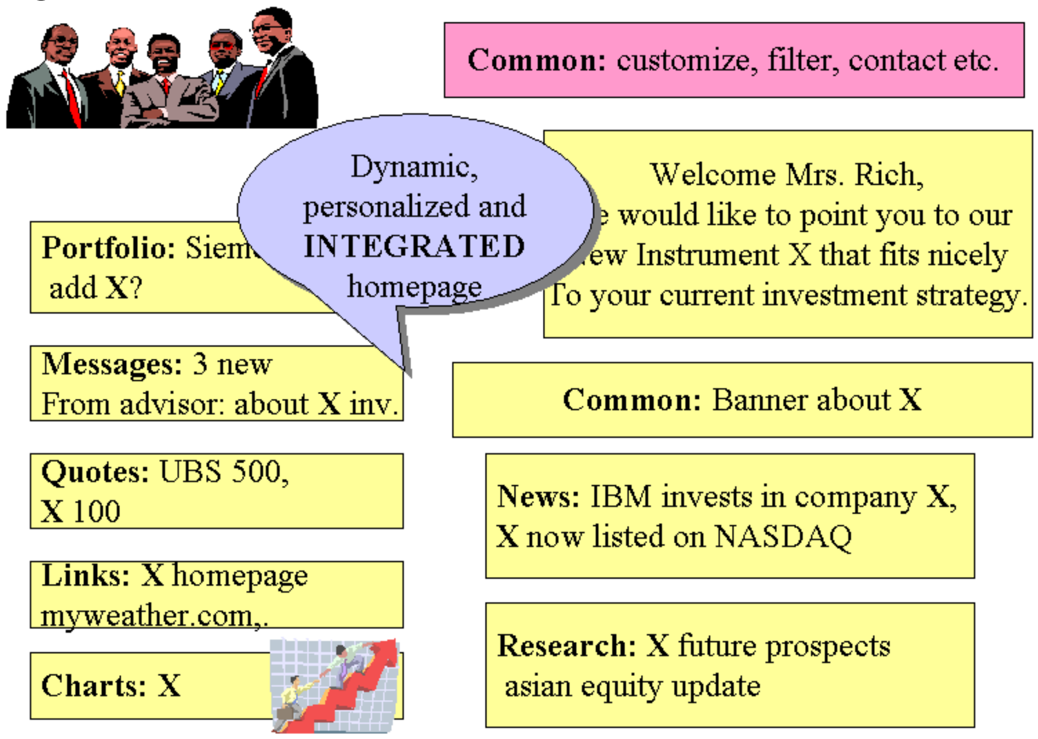
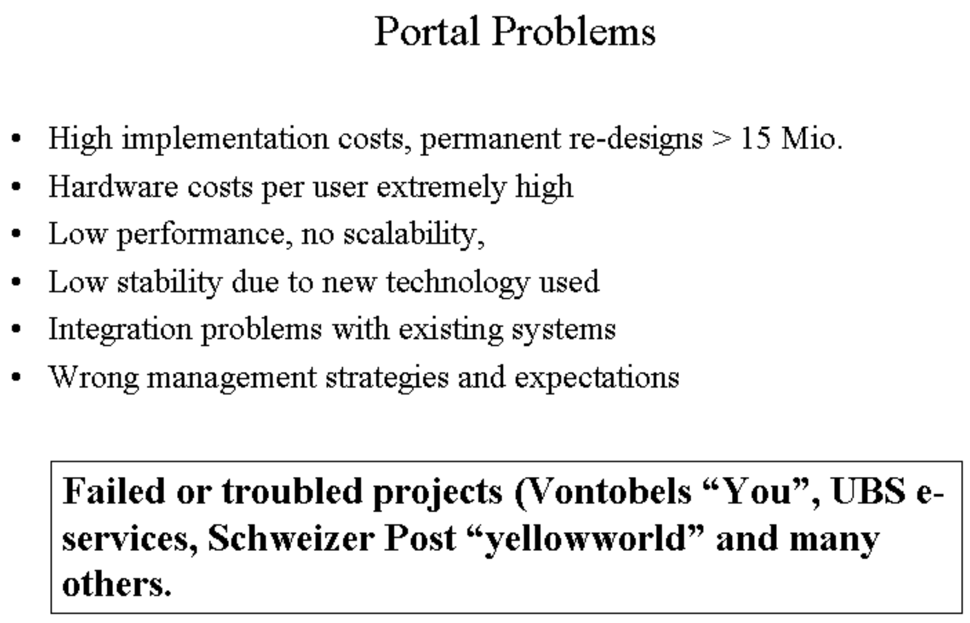

Common: customize, filter, contact etc.
Dynamic and personalized homepage
Welcome Mrs. Rich, e would like to point you to our New Instrument X that fits nicely To your current investment strategy.
Portfolio: Siemens, Swisskom, Esso,
Messages: 3 new From foo: hi Mrs. Rich
Common: Banner
Quotes: UBS 500, ARBA 200
News: IBM invests in company Y
Links: myweather.com, UBS glossary etc.
Research: asian equity update
Charts: Sony

Figure 2.3.


Common: customize, filter, contact etc.
Dynamic, personalized and INTEGRATED homepage
Welcome Mrs. Rich, e would like to point you to our ew Instrument X that fits nicely To your current investment strategy.
Portfolio: Siem add X?
Messages: 3 new From advisor: about X inv.
Common: Banner about X
Quotes: UBS 500, X 100
News: IBM invests in company X, X now listed on NASDAQ
Links: X homepage myweather.com,.
Research: X future prospects asian equity update
Charts: X

Problems

Figure 2.4.



Portal Problems

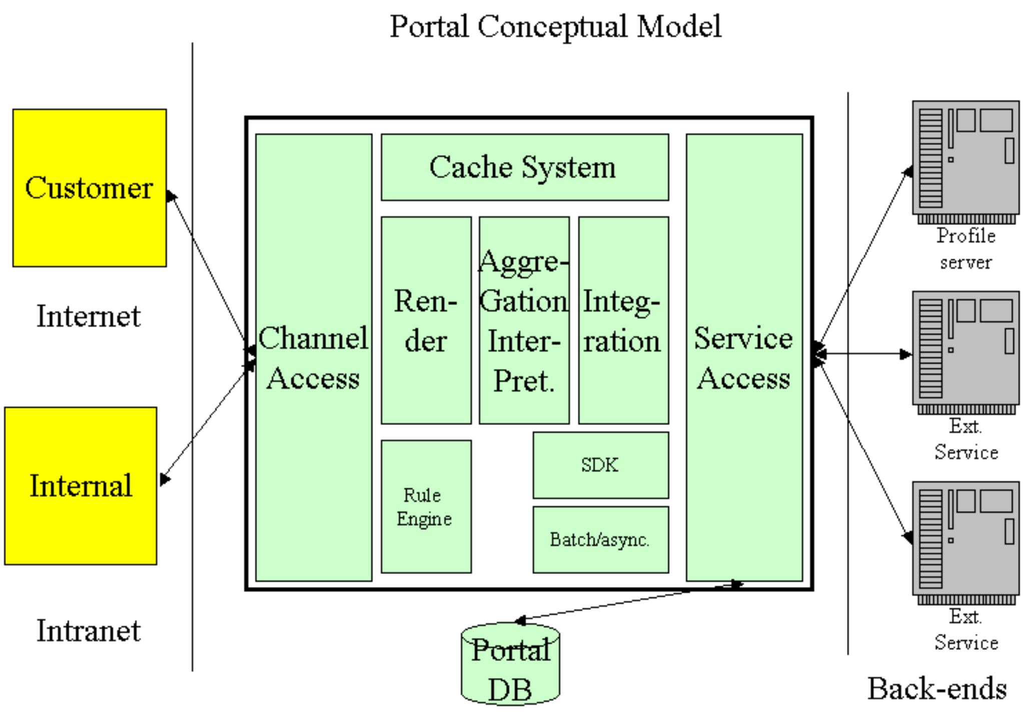
- High implementation costs, permanent re-designs > 15 Mio.
- Hardware costs per user extremely high
- Low performance, no scalability,
- Low stability due to new technology used
- Integration problems with existing systems
- Wrong management strategies and expectations

Failed or troubled projects (Vontobels “You”, UBS e-services, Schweizer Post “yellowworld” and many others.

Chapter 3. Portal Conceptual Model

Building Blocks

Figure 3.1.



Model2 Architecture

Figure 3.2.

Simple Model2 Architecture

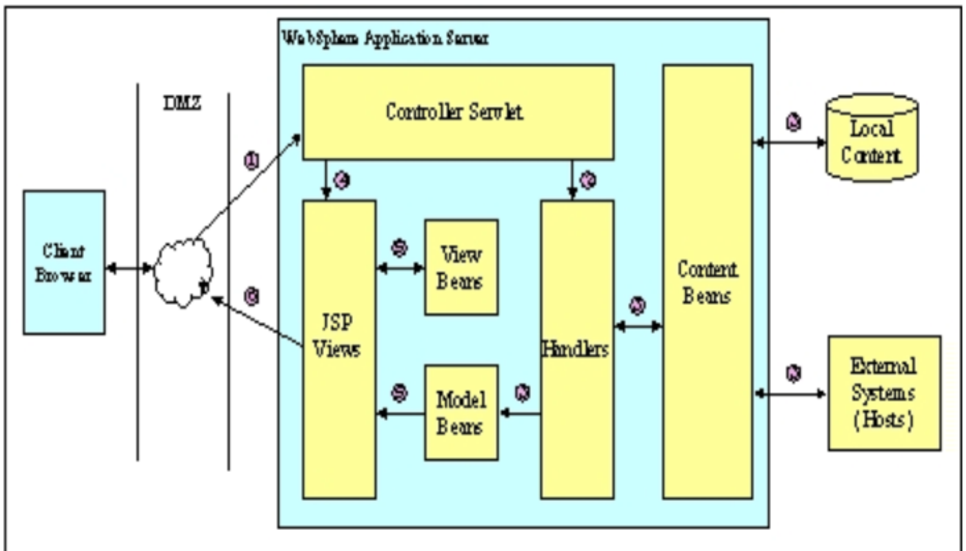


Figure 1, Technical Architecture

Request processing

To get a better understanding of what really happens during a page request (and as an anti-dote for all the fancy sounding Java-isms) I found the following paragraph from Jon S. Stevens quite enlightening: (Please replace "Turbine" with "AEPortal" – same stuff here).

The current encouragement that Turbine gives to developers is to do a mapping between one Screen (Java) and one Template (WM or V). The way it works is that you build up a Context object that is essentially a Hashtable that contains all of the data that is required to render a particular template. Within the template, you refer to that data in order to format it for display. I will refer to this as the "Push MVC Model." This IMHO is a perfectly acceptable, easy to understand and implement approach.

We have solved the problem of being able to display the data on a template without requiring the engineer to make modifications to the Java code. In other words, you can modify the look and feel of the overall website without the requirement of having a Java engineer present to make the changes. This is a very good step forward. However, it has a shortcoming in that it makes it more difficult to allow the template designer (ie: a non programmer) the ability to move information on a template from one template to another template because it would require the logic in the Java code to be modified as well. For example, say you have a set of "wizard" type screens and you want to change the order of execution of those screens or even the order of the fields on those screens. In order to do so, you can't simply change the Next/Back links you need to also change the Java code.

So what does this mean for a JSP developer who wants to create a new view on some information?

1) Find out what handlers will create what pieces of information ("models" in AEPortal terms). Do this by looking at the handler(s) code.

2) If you need pieces of information from several handlers, have someone create a handler that calls

all the needed handlers internally.

3) Go into the source code of the handler(s) and write down the keys used to store the pieces of information into the request hashtable.

4) Extract the information in the JSP.

This is a cumbersome and error-prone process and it requires programming skills at all levels. The "re-use" of handlers creates subtle dependencies. There is NO definition of page content or information.

The whole thing turned upside down would look like this:

"Instead of the developer telling the designer what Context names to use for each and every screen, there is instead a set of a few objects available for the template designer to pick and choose from. These objects will provide methods to access the underlying information either from the database or from the previously submitted form information."

But this would require us to have a level beyond the (procedural) "handlers" where definitions of pages and page content, fragments and parts of information exist. This in turn would allow us to build editors even for dynamic and personalized pages.

But an even more important side-effect is that this architecture would support information fragments much better. And fragments are the basic building blocks for caches and allow the caching even of highly dynamic and personalized pages. This will be explained further down.

And last but not least: Andreas Kapp pointed me to another possibility: to not only cache those fragments but to make them persistent too – on a per user basis. This means that the personalization decisions are kind of frozen within a personal fragment and there is no need to always re-calculate the filters, selections etc. permanently during each request. Only the content that really changed needs to be filled in.

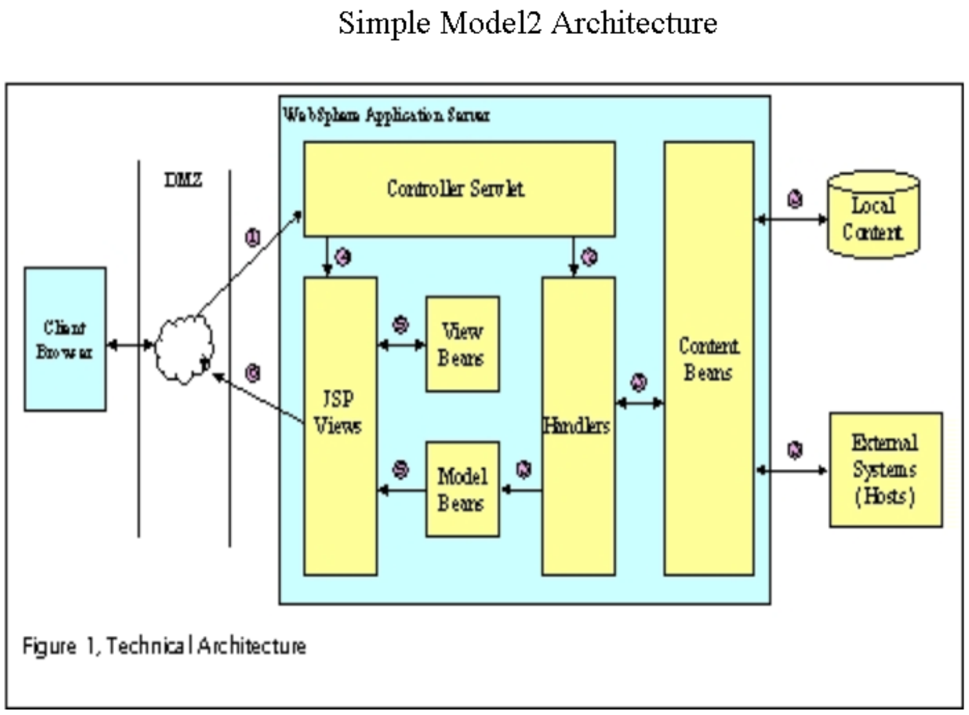
We will discuss the requirements of persistent, self-contained fragments in the chapter on fragments.

Chapter 4. Reliability

Simple page processing

The diagram below – taken from the Infrastructure Overview - shows the typical flow of control for a simple page request in a J2EE Model 2 Architecture.

Figure 4.1.



1. POST/GET. The request is submitted from the client browser and arrives at the Controller servlet. There is a single instance of the Controller servlet per web application. The Controller multi-threads, processing multiple client request concurrently. All requests for the web application arrive at the Controller.
2. Dispatch. The controller determines who the requesting user is, determines which page they are requesting, sets up the necessary context and invokes the correct *Handler* to process that request.
3. Create/Update. There is typically one Business Logic Handler per page. The handler is responsible for initiating the business operations requested by the user. Typically this will involve interaction with backend systems to retrieve or modify some persistent state. When the processing has completed, the Handler is responsible for creating or modifying State Data (the Model) held in the Web Application to represent the results. The Handler then completes and control passes back to the Controller.
4. Forward. The Controller will then determine which JSP to invoke to display the results. Normally the JSP is determined automatically based on the Requested page, however the Handler

may have over-ridden the default if it wishes.

5. Extract. The role of the JSP is to render the page as HTML. When the JSP needs to display application data it extracts it directly from the State Data (Model) that has previously been setup by the Handler.
6. Respond. When the JSP has finished rendering the page it is returned to the client browser for display.§

Therefore, each page to be delivered to the client typically involves writing a triplet comprising: a Handler to process the Business Logic; a Model to hold the result data; a JSP to display the results back to the client.

AEPortal uses the JADE infrastructure to support the above process.

RAS properties

The processing of a simple page request has the following RAS properties:

- No additional threads created. Handler uses servlet thread
- During the request the AEPortal database is contacted (e.g. for profile information) and optionally an external service is accessed (e.g. to load a research document from a web-server
- Processing is sequential and response-time constrained: the handler cannot use wait times (e.g. waiting for network responses) for other tasks
- If the handler is blocked, the whole request coming from the web container is blocked too. If the maximum number of open connections is reached, no new request can enter AEPortal WHILE the handler(s) are busy.
- No timeouts are specified for a handler. If timeouts happen they do so within the external service access API.
- There is currently no external service access API that would offer a Quality-of-Service interface e.g. to set timeouts or inquire the status of a service.

Behavior in case of failures

Let’s assume that an external service becomes unavailable. Eventually a handler waiting for this resource will get a timeout in the access API of that service and return with an error. This may take x seconds to become effective. (We need to know more about timeouts in our access APIs).

While waiting for the external resource a handler will hold on to some system resources but at the same time block an entry into the system. The effects on the system resources should be benign.

The user will have to wait until the timeout happens and the request returns. A different user with a request to a different resource will not be affected but a user going after the same resource will see the same delay while waiting for a timeout.

It would be an improvement for both system and user if we could tag a service as being unavailable and not start any new requests against this service. But this raises a couple of questions:

- Who turns a service off? Is it the service itself? A handler?

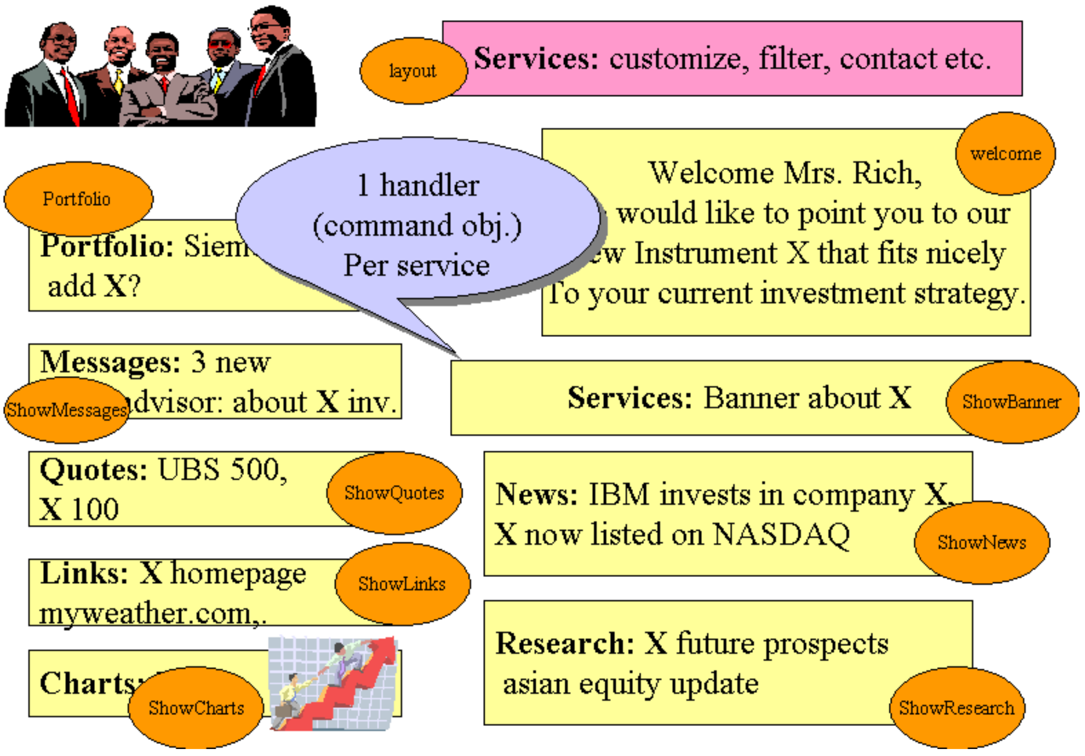
- When is a service turned off? If it does not answer at all? If it is simply slower than usual? What is too slow?
- When is a service turned on again? After a certain time or number of requests?
- Who can turn on a service again?

For now we need to make sure that the timeouts hidden in our access API's are short enough to avoid blocking too many requests too long.

Debug has shown some requests waiting 4 minutes or more e.g. on Quotes. This a severe system drain and a bad user experience.

Multi-Page (parallel) processing

Figure 4.2.


layout
Services: customize, filter, contact etc.
Portfolio
Portfolio: Siem
add X?
1 handler (command obj.) Per service
welcome
Welcome Mrs. Rich, would like to point you to our ew Instrument X that fits nicely To your current investment strategy.
Messages: 3 new
ShowMessages
dvisor: about X inv.
Services: Banner about X
ShowBanner
Quotes: UBS 500, X 100
ShowQuotes
News: IBM invests in company X, X now listed on NASDAQ
ShowNews
Links: X homepage myweather.com,.
ShowLinks
Research: X future prospects asian equity update
ShowResearch
Charts:
ShowCharts

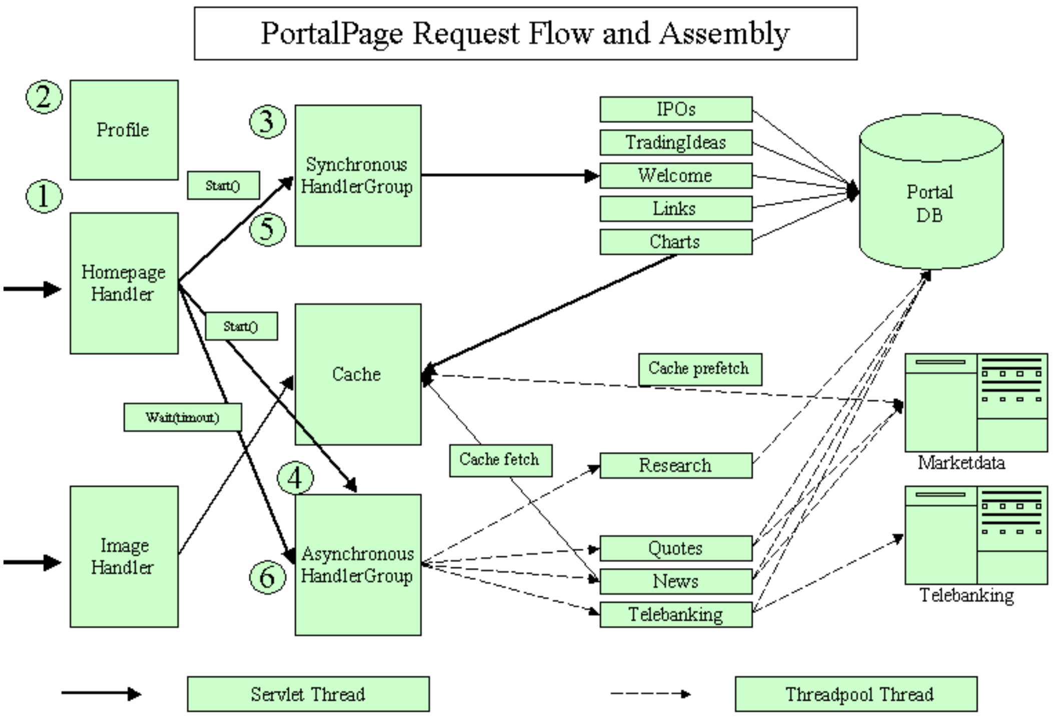
The homepage of AEPortal is called a "Multi-Page" because it combines several otherwise independent services on one page. Initially the internal processing of the homepage happened in the same way as for any other page: a handler running on a server thread (i.e. the thread that comes from the web container) collects the information and forwards the results to the view (via the controller).

It soon turned out that the AEPortal homepage had special requirements that were not easily met by the standard page processing:

- Several external services needed to be contacted for one homepage request
- Large waits on external I/O
- Every additional homepage service added to the overall request time (= the sum of all individual processing times)
- Rendering could not start until all services had finished. Frames were not allowed and therefore no partial rendering possible. Users did see nothing while waiting for the whole page to complete
- In case of a service failure (e.g. news) not only individual page requests for news would block – almost every homepage would be affected too because news are a part of many personal homepage configurations
- A simple page request blocking holds on to a small number of system resources. A homepage request blocking would hold on to a much larger number of system resources – potentially causing large drops in available memory. (Something we have observed during Garbage Collection debug)

While processing of multi-pages would have to be somewhat different there was a necessity to reuse the existing handlers and handler architecture because of tight deadlines. This lead to the following architecture:

Figure 4.3.



The processing steps 1 – 3 from above apply here as well. The homepage handler is a regular handler like any other.

1. Using the servlet thread, the controller forwards the incoming request to the homepage Handler.

2. Homepage handler reads profile information to select which services of the homepage need to run for this user (access rights are of course respected as well)

3. Every service has a page description in ControllerConfig.xml telling the system about the necessary access token, handler names and last but not least which data sources the service will use. Depending on the data sources (available are: AEPortalDB, OTHERDB, HTTP) the handler for this service will be put in one of two HandlerGroup objects: Handlers using only the AEPortalDB are put in the synchronous handler group. Handlers using HTTP sources end up in the asynchronous handler group.

4. Homepage handler calls start() on the asynchronous group first. It iterates over all available handlers and starts every one with a thread from the AEPortal threadpool.

The behavior of the threadpool needs to be checked. Will every new request be put into a large queue or will the requester block on adding to the queue if all threads are busy? This will affect system behavior on a loaded system.

1. Then homepage handler calls start() on the synchronous group. Again it iterates over all available handlers but uses its own thread (which is the servlet thread) to execute the handlers sequentially.

2. After executing all handlers in the synchronous group homepage handler calls wait(timeout) on the asynchronous group. The timeout is configurable (currently 30 seconds). When all handlers are finished OR the timeout has happened, homepage handler returns back to Controller.

The assumptions behind these two groups are as follows:

- handlers in the synchronous group are both fast and reliable (because they only talk to the AE-Portal database). They could run within their own threads as well but the overhead for the increased thread context switches are not worth the effort. We see execution times of less than one second for the whole synchronous group.

- The reasons for handlers to go into the asynchronous group are more diverse. The first and obvious reason is that the handlers experience large waiting times on I/O because they access e.g. other web servers. By starting a request as soon as possible we can effectively run several in parallel. Doing so we avoid the overall homepage request time to be the sum of all single handler times. Tests have shown significant savings here. To achieve this effect the wait() method would not need a parameter timeout.

- The timeout parameter is available because external services not only cause I/O waiting times but are also sometimes unreliable too. The assumption was that a full homepage request should not have to wait for a single handler simply because its associated external service is very slow or even unavailable. While this is a valid requirement the consequences of the timeout parameter are much more complicated than initially thought. This will be discussed below.

- (Research, Telebanking and Charts are currently special cases. Research does NOT use HTTP access for the homepage part but is currently fairly slow and is therefore in the asynchronous group. This will change soon and it will move to the synchronous group. Telebanking is different in development and production and it is currently not clear how fast and reliable access to its external services will be in production. This is why it is in the asynchronous group even though its runtime in development is only around 50 ms. The charts part of the homepage is worth an extra paragraph below.

Homepage-charts service

Charts looks like a service that would need an external service (External Data System) for its pictures – and it does so, just not within the charts homepage request.

The homepage request of charts uses the asynchronous requester capabilities of the AEPortal cache and simply requests certain pictures to be downloaded from External Data System (if not yet in the cache).

Charts Handler schedules the request synchronously but the request itself runs in its own thread in the background, taking a thread from the AEPortal threadpool.

The homepage only contains the URLs of the requested images. During rendering of the homepage the browser will request the images asynchronously by going through the AEPortal image handler.

The fact that charts causes a background thread to run is important in case of external system failure: what happens if the charts external service is down? Or under maintenance? Currently the background thread that is actually trying to download the image will get a timeout after a while and finish. In the meantime an image request from the browser will wait on the cache without a timeout. What happens in case of a failure? Will there be a null object in the cache? Anyway – the image request blocking without timeout is not such a big problem because it runs on a servlet thread and therefore blocks an entry channel into AEPortal at the same time – no danger of AEPortal getting overrun.

The background thread of the asynchronous requester is more dangerous because a failure in an external service can lead to all threads of the threadpool being allocated for the cache loader.

Threads should NOT die and if they do the threadpool needs to generate a new one.

Failure behavior needs to cover maintenance periods of external services too, especially the single point of failure we got through MADIS (news, quotes and charts all rely on MADIS)

Note: The image cache cannot get too large! Should the images be cached in the file system instead?

RAS properties without timeout

The processing of a multi-page request has the following RAS properties:

- Additional threads are used. Handler uses server thread to start the threads. At least one thread must be available (possibly after waiting for it). A homepage request cannot work with a threadpool that does not have any threads and it cannot use the server thread instead (which would be equivalent to putting all necessary handlers into the synchronous group). There are no means to check for available threads in the threadpool.

- The size of the threadpool is still an open question. Testing has shown that an average homepage request uses between two to four threads but his was before the introduction of the separate handler groups. In addition to this the asynchronous requester capability of the cache (see below) will need threads too.

- It is still an open question whether the threadpool should have an upper limit. One of the problems associated with no upper limit is that currently the threadpool does not shrink – it only grows till the max. value is reached. The question of upper limits is much more difficult to answer for the case of timeouts being used (see below)

- During the request the AEPortal database is contacted many times (e.g. for profile information) and parallel to this external services are accessed. There is no common transaction context between the handlers.

- Processing is sequential as well as parallel: The best case would be if the runtime of the synchronous block is equal to the runtime of the asynchronous handler group (which is started first). Testing has shown that in many cases the external services are slower (by a factor of 2 – 5)
- A single slow service leads to a considerable system load since the other handlers have already allocated a lot of memory.
- If one of the handlers involved blocks the whole request coming from the web container is blocked. If the maximum number of open connections is reached, no new request can enter AEPortal WHILE the handler(s) are busy. This is also the case if one of the handlers in the asynchronous group blocks because the homepage handler itself will wait for the whole group to finish (in case of no timeout set)
- No timeouts are specified for a handler. If timeouts happen they do so within the external service access API.
- There is currently no external service access API that would offer a Quality-of-Service interface e.g. to set timeouts or inquire the status of a service.
- The current policy for error messages is as follows: A failure in a service API should not make the JSP crash because of null pointers. The JSP on the other hand cannot create a specific and useful error message because it is not informed about service API problems. This is certainly something that needs to be fixed in a re-design.

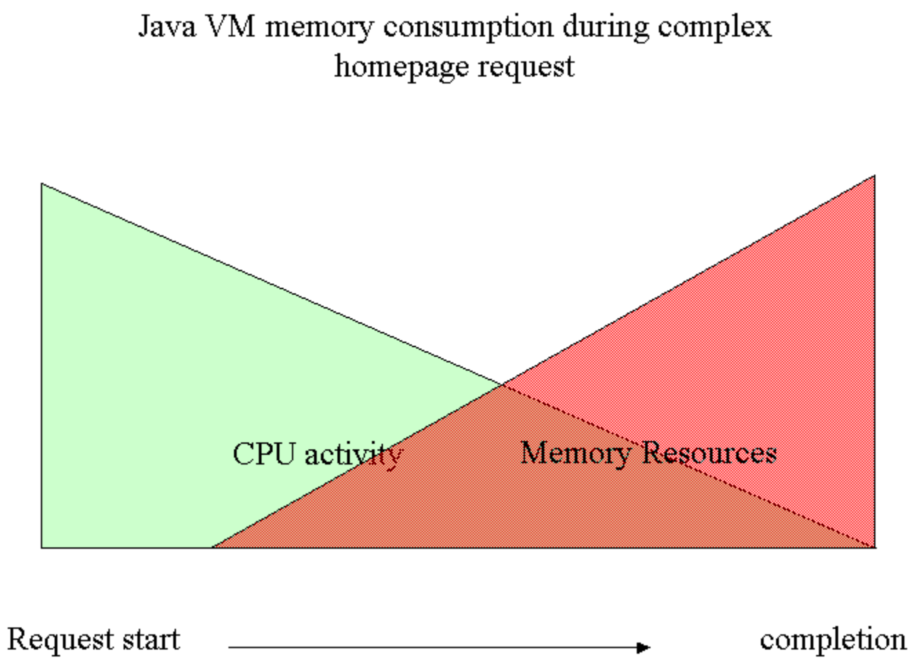
Behavior in case of failures

Again, just like the in the case of a simple page request, let’s assume that an external service becomes unavailable. Eventually a handler waiting for this resource will get a timeout in the access API of that service and return with an error. This may take x seconds to become effective. (We need to know more about timeouts in our access APIs).

While waiting for the external resource ALL HOMEPAGE HANDLERS that belong to one request will hold on to some system resources. And so will ALL HOMEPAGE REQUESTS do which include this special handler. On top of that - three of the homepage services go to ONE external service (MADIS) and all external services are subject to change (hopefully with an early enough warning to AEPortal). This means that a blocked homepage request has a much bigger impact on system resources than a simple page request. Actually, without the ability to close down a specific service, any interface change in an external service could bring AEPortal down easily.

And since the homepage is at the same time the most important als well as the first service after login, a blocking service from the homepage is a critical condition that can quickly drain the system of its memory resources.

Figure 4.4.



The good news: The request blocks an entry into the system (servlet engine) and prevents overrun.

The effects on the user are also different compared to a simple page request: If a simple request hangs the user can always go back to the homepage and chose a different service. This is not possible if the homepage hangs. We do not offer horizontal navigation yet (going from one service directly to any other service). That means that with a blocking homepage a user gets NOTHING. From an acceptance point of view a quick and reliable homepage is also a must.

The dire effects of a failure in a homepage service have led to the introduction of a timeout for waits on the asynchronous service group. The requirements and consequences of a timeout will be discussed next.

RAS properties with timeout

After the things said above it should be clear that the external services make AEPortal very vulnerable, especially the most important portal page. Before we dive into the implementation of timeouts a speciality of the Infrastructure architecture needs to be explained: the relation between handler, model and jsp.

Note: the "models" should really be result objects. The JSP should be a fairly simple render mechanism that can ALWAYS rely on result objects to be present – even in case of service API errors. Right now our JSPs are overloaded with error checking code.

Handlers create model objects. If a handler experiences a problem it can store an exception inside a model object and make it accessible for the jsp. But what happens during a multi-page request? The first thing to notice is that the Homepage jsp cannot expect to find a model object for every possible homepage handler. The user may not have the rights for a certain service or has perhaps deconfigured it. In these cases the handlers do not run and therefore do not create model objects.

The introduction of a timeout while waiting for asynchronous handlers offers another chance for "no model". A handler blocks on an external service but the homepage handlers times out and returns to the controller and finally to the jsp. The handler did not create or store a model object yet. The only way around this problem is that the homepage handler gathers statistics about the handlers from the handlergroup and stores it in the Homepage-model. Now the jsp can learn about which handlers returned successfully and which ones didn't.

The implementation of a timeout is simple: the homepage handler calls waitForAll(timeout) on the handler group and returns either because all handlers of the group signaled completion to the group or because of the timeout. The consequences are much more difficult. What happens to the handler that did not return on time? The answer is simple: Nothing. It is very important to understand that the thread running the handler is not killed. Killing a thread in Java is deprecated and for a good reason too. When a thread is killed all locks held by this thread are immediately released and if the thread was just doing a critical operation on some objects state, the object might be left in an inconsistent state.

This has an interesting effect on system resources: The homepage handler will return and after rendering the request will return back to the servlet engine and by doing so free an input connection into the engine – WHILE THERE IS STILL A THREAD RUNNING ON BEHALF OF THIS REQUEST WITHIN AEPortal. A new request can enter the system immediately, will probably hit the same problem in the handler that timed out in the previous request and return – again leaving a thread running within AEPortal. Of course, these threads will eventually time out on the service API and return to the pool but on a busy system it could easily happen that we run out of threads for new homepage requests (or even asynchronous requests on the cache)

Does it help to leave the upper bound of the threadpool open? Not really since requests could come in so fast that we would exhaust any reasonable number of threads. And remember – we can't shrink this number afterwards.

Note: we have discussed an alternative: Wouldn't it be better to crash the VM through an exploding threadpool? In this case the websphere system management would re-start the application server!

If we could prevent the new request from running the problematic handler we could avoid losing another thread. But this would require the functionality to disable and enable services.

Without being able to disable and enable services automatically (basically a problem detection algorithm) a timeout does not really make sense. It is even dangerous if set too low.

Effect of caching on the scheduler

The handler threading mechanism was introduced at a time when many handlers used little or no caching at all. It could be expected that every handler in the asynchronous group would have to go out on the network and request data.

The membership in the asynchronous group therefore became a property of the handler – tagged onto the page description. While this is easily changed by a change to the configuration file it still presents a problem in case of advanced caching: If the data the handler has to collect is already in the cache, the handler will return almost immediately. This is a waste of computing resources because the overhead of thread scheduling is bigger than the time spent in the handler to return the CACHED data.

The homepage handler on the other side could not know in advance if e.g. the news handler will find the requested data in the cache or not. It needs to start the news handler in its own thread (per configuration) even if the news handler will run only 10 ms.

Why doesn't the homepage Handler know which data the news handler will retrieve? Because there is no description of those data available! The only instances that know about certain model objects are handlers and their views. This makes caching and scheduling much harder.

A design flaw:

What looks like a little nuisance hides a much bigger design flaw in the portal architecture: The architecture is procedure/request/transaction driven and not data/publishing driven:

- Without running a handler no results ("models") come to exist
- The results (models) do not survive a request
- Only the code within a handler knows what data ("models") will be created and where they will stored (and how they will be called)
- A page does not have a definition of its content – quite the opposite is true: a handler defines implicitly what a page really is. Example: the content of the homepage is defined as the set of handlers that need to run to create certain data ("models")
- Handlers need to deal with caching issues directly and internally. No intermediate layer could deal with cached data because nobody besides each specific handler knows which data should/could be cached
- The framework maps GET and POST request into one request type – negating the different semantics of both request types in http. There seem to be no rules within our team with respect to the use of GET or POST.

We have already seen some of the consequences of this approach with respect to caching and scheduling. But think about using the AEPortal engine behind a new access channel or just AEPortal light. Whoever wants to extract information through AEPortal needs to know about handlers and their internals (models). Instead of knowing about data and data fragments, clients need to call handlers. Again, this architecture is good for transactional purposes but it is disastrous for publishing purposes.

A few hints on a data-driven alternative:

1. A homepage request comes in.
2. The system retrieves the homepage description containing links to fragments.
3. The systems tries to retrieve the fragments from the caching layer, with the user context as a parameter
4. If the cache contains the item in the version needed (we have personalized data!) it is returned immediately.
5. If we have a cache miss, the proper requester turns around and gets a handler to retrieve the data (synchronously or asynchronously)
6. The items returned are assembled and forwarded to the proper view.

This means for clients that they do not know about system internals. All they need to know is the INFORMATION they want – not which handlers they have to call to retrieve an information set that hopefully contains all the data they need.

We will discuss this some more in the chapter on caching, when we meet the problem of document fragments.

For more information about this have a look at:

http://www.apache.org/turbine/pullmodel.html [http://www.apache.org/turbine/pullmodel.html]

Its focus is on GUI flexibility – UI designers cannot change the GUI without also changing the Java based handlers that create the result objects - but the reasoning also applies to caching and fragments.

Another design flaw

The current handler design did not force developers to separate different use-cases into different handlers. A typical example is the authentication handler providing the rendering on behalf of the authentication front-end. Different requests all end up in one handler – only distinguished by different parameters. This has the following problems associated:

- It is hard to distinguish read-only from change operations
- Caching becomes very hard because the validator needs to know the different parameters of the requests
- Access control no longer works on page/fragment level. It has to happen context sensitive within the handler code – thereby depending on the developer.

The fragment based architecture described below needs to separate the different requests into clearly distinguishable fragments.

The handlers need a more generic design allowing the configuration of e.g. a fragment handler by specifying the service needed etc. This is e.g. done in the Portlet approach (Apache Jetspeed)

Multiple Multi-Pages

The issue of multiple multi-pages or homepages is caused by the simple fact that our current homepage cannot grow endlessly – we could not delivery all those fragments in a reasonable time. Not to mention that the whole page might become very confusing.

Technically the problem is not very hard to solve:

1. the homepage handler needs to be generalized into a multi-page handler. It looks at the requested page name from RequestContext and retrieves the proper page description
2. HandlerGroup and derived synchronous and asynchronous handler groups would need to be generalized a bit.
3. The page descriptions in the ControllerConfig.xml file need to be extended to allow page links within page elements (in case a service appears in several multi-pages
4. A navigation scheme between the multi-pages would be necessary. But this would be the same as for horizontal navigation needed anyway.
5. Some rules need to control the number and selection of services per multi-page

Service Access and Reliability

The way external services were accessed proved to be the most influential factor for system reliability. What was missing was a service API through which all access had to go and which would be a major point for monitoring those external services. The chapter on portal architecture will show what is needed.

- A Distribution Architecture defining the characteristics of external services (response time, downtime etc.)
- A Service Access Layer that shields the portal from service problems and allows service maintenance.

Chapter 5. Performance

Caching

The end-to-end dynamics of Web page retrievals

A large-scale web application needs to apply an end-to-end view on how pages are created and served: client side and server side.

This covers issues like dynamic page creation, the structure of pages, compression and load-balancing issues and goes from http headers settings over colors and page structure right down into application architecture.

We will talk about the current settings for client side caching in our infrastructure shortly and then move on to server side caching.

Cache invalidation

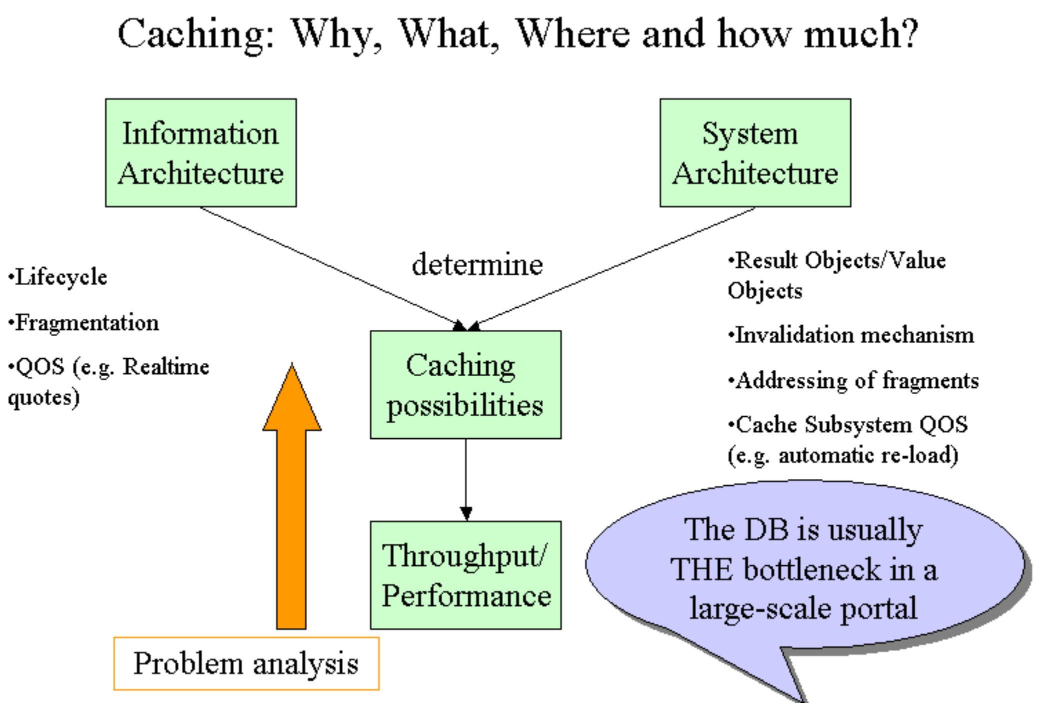
The biggest problem in caching dynamic and personalized data is cache invalidation. Client side browser caches as well as intermediate proxy caches cannot be forced to invalidate an entry ON DEMAND. These caches either cache not at all or for a certain time only. The newer HTTP1.1 protocol also allows them to re-validate a fragment by going to the server – driven by cache-control settings for the page.

The result is that cache-control settings for browser and proxy caches need to be conservative.

Server side caches on the other side MUST HAVE AN INVALIDATION INTERFACE and possibly also validator objects that decide about when and if a certain fragment needs to be invalidated.

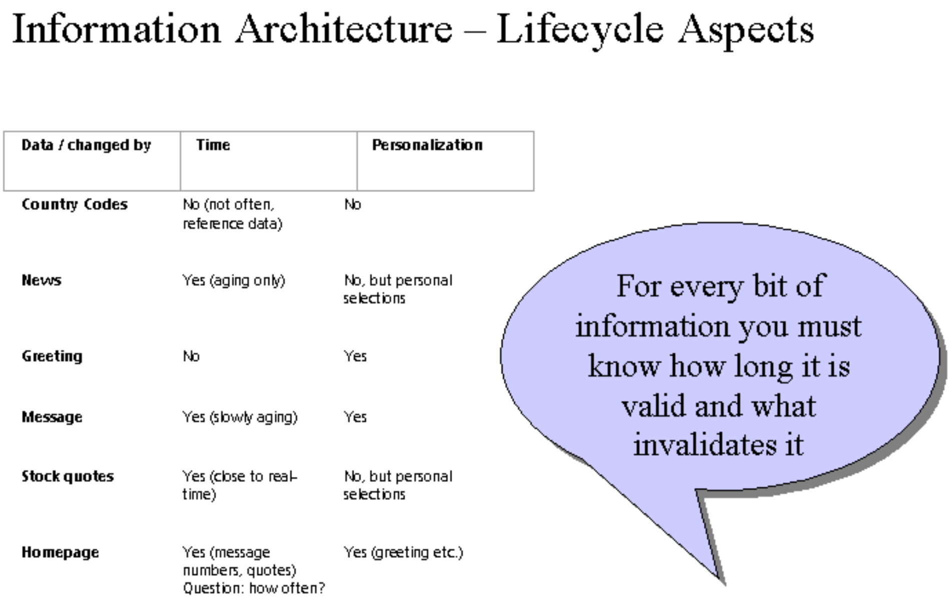
Information Architecture and Caching

Figure 5.1.


Caching: Why, What, Where and how much?
Information Architecture
System Architecture
determine
•Lifecycle
•Fragmentation
•QOS (e.g. Realtime quotes)
•Result Objects/Value Objects
•Invalidation mechanism
•Addressing of fragments
•Cache Subsystem QOS (e.g. automatic re-load)
Caching possibilities
Throughput/ Performance
Problem analysis
The DB is usually THE bottleneck in a large-scale portal

Tagging the portal information with respect to its lifetime is a necessity:

Figure 5.2.



Information Architecture – Lifecycle Aspects

Data / changed by	Time	Personalization
Country Codes	No (not often, reference data)	No
News	Yes (aging only)	No, but personal selections
Greeting	No	Yes
Message	Yes (slowly aging)	Yes
Stock quotes	Yes (close to real-time)	No, but personal selections
Homepage	Yes (message numbers, quotes) Question: how often?	Yes (greeting etc.)

For every bit of information you must know how long it is valid and what invalidates it

Client Side Caching

AEPortal being a personalized service our initial approach to client side caching was to simply turn it off completely.

private static String sExpiresValue = "0";

private static String sCacheControlValue = "no-cache, no-store, max-age=0, s-maxage=0, must-revalidate, proxy-revalidate"; // HTTP 1.1: do not cache nor store on proxy server. AKP

private static String sPragmaValue = "no-cache";

These values are currently set at the beginning of the service method of our controller servlet. They are the same for all pages. It would not be hard to make them page specific – driven by a tag in our ControllerConfig.xml. That’s what e.g. the struts package from Apache.org wants to do in the next release.

We do not use a "validator", e.g. LAST_MODIFIED which means that clients will not ask us to validate a request. Instead they will always pull down a fresh page.

We also do not use the servlet method getLastModified() which has the following use case:

It's a standard method from HttpServlet that a servlet can implement to return when its content last changed. Servers traditionally use this information to support "Conditional GET" requests that maximize the usefulness of browser caches. When a client requests a page they've seen before and have in the browser cache, the server can check the servlet's last modified time and (if the page hasn't changed from the version in the browser cache) the server can return an SC_NOT_MODIFIED response instead of sending the page again. See Chapter 3 of "Java Servlet Programming" for a detailed description of this process.

Jason Hunter, http://www.servlets.com/soapbox/freecache.html [http://www.servlets.com/soapbox/freecache.html]

A simple example that a personalized homepage need not exclude the use of client side caching:

If the decision to use the cached homepage can be based purely on the age of the homepage (e.g. 30 secs.) the getLastModified() would simply compare the creation time of the homepage (stored in session?) with the current time.

This would help in all those re-size cases (Netscape). It would also decrease system load during navigation (we don’t have a horizontal navigation yet).

Please note: We are talking the full, personalized homepage here. Further down in "client side caching" we will also take the homepage apart – following an idea of Markus-A.Meier.

Results and problems with client side caching

First the controller servlet was changed to set the EXPIRES header and the MAX_AGE cache control value both to 20 seconds default per page. To enable the getLastModified() mechanism a validator (in our case LAST_MODIFIED) was set to the current time when a page was created. And getLastModified() returned currenttime-20000 by default. No explicit invalidation of pages was done.

Note: the controller servlet was modified in several ways:

It now implements the service method, overriding the one inherited from HttpServlet. I noticed that this is the method that seems to call getLastModified() – allowing us to distinguish the case where getLastModified() is called to set the modification time vs. it being called to test for expiration (see J.Hunter)

It is unclear if overriding the service() method is actually a no-no.

The servlet now also implements the destroy() method – even if it only logs an error message because right now we are not able to re-start the application (servlet) without a re-start of the application server. This is because we use static singletons. The destroy method should at least close the threadpool and the reference data manager.

It is unclear under which circumstances the websphere container would really call destroy(). Could our servlet and container experts please comment on this?

We did not set the MUST_VALIDATE header yet but some pages would probably benefit from doing so.

Problems:

- the expiration time need not only depend on the page. Different users could possibly have a different QOS agreement for the same pages. Real-time quotes are a typical example. Either we use different pages for those customers (could force us to use many different pages) or we can specify an array of expiration times per page
- After changing the homepage layout (myprofile), a stale homepage containing the old services was served once to the user. We need to use MUST-VALIDATE and a better handling of the getLastModified() method.
- We don’t know if business will authorize an expiration time of 20 seconds for every service.
- We don’t know how clients will use our site. (Our user interface and usability specialist Andy Binggeli has long since requested user acceptance tests) and therefore we must guess usage patterns, e.g. navigational patterns

Results:

- Three out of four homepage requests for one user came from the local browser cache
- Navigation between the homepage and single services was much quicker
- Browsers treat the "re-load" button differently, e.g. Opera requests an uncached page when the re-load button is hit. Netscape needs a "shift + re-load" for this.
- Javascript files seem to get no caching, at least within our test-environment. This would mean a major performance hit as some of them are around 60k big.

Note

Check on Javascript caching in production!

While client side caching will not affect our load tests (e.g. login, homepage, single-service, logout) regular work with AEPortal would benefit a lot.

A possible extension of the page element that covers the content lifecycle could be like this:

Example 5.1. Lifecycle definitions for cachable information objects

```
<!ELEMENT page (...,lifecycle ?,..) >
< !ATTLIST page
        -- refer to a named lifecycle instance by idref (optional)--
lifeCycleRef idref #implied >
< !ELEMENT lifecycle (#empty) >
<!ATTLIST lifecycle
        -- allows to refer to a certain lifecycle definition
name ID #implied
        -- after x milliseconds the user agent should invalidate the page.
The system will assume a reasonable default if none given. A zero will
tell the user agent to NOT cache at all --
expires CDATA #implied
         -- the user agent should ask server after expiration time --
askAfterExpiration (yes|no) yes
        -- the user agent should ALWAYS ask for validation --
askAlways (true|false) false
        -- the system will ask the given validator for validation during
a getLastModified() request OR when a validator (LAST_MODIFIED or
ETAG needs to be created. Allows pages to specialize this --
validator CDATA #implied >
         -- experimental: what to do in case the backend is down: --
useCachedOnError (y|n) n >
```

Note

The lifecycle element is an architectural element. It is intended to be used in different contexts e.g. pages, page fragments etc. Therefore a lifecycle instance can have a name that serves as an ID. Users of this instance can simply refer to it and "inherit" its values.

This does not prevent users from specifying their own lifecycle instance and STILL refer to another one. In this case the users own instance will override the one that was referred to.

Proxy Side Caching

AEPortal includes a number of static images that should be served from the reverse proxies. The same is true of our Javascript files.

Note: who in production will take care of that?

Further caching of information is a tricky topic because the information might be personalized. We do not allow proxy side caching right now. But for some information it might be OK to use the public cache-control header.

This chapter obviously needs a more careful treatment.

(Image) Caching and Encryption

"Encryption consumes significant CPU cycles and should only be used for confidential information; many Web sites use encryption for nonessential information such as all the image files included in a Web page" (23).

Is there a way to exclude images from encryption within an SSL session?

Can we cache encrypted objects (e.g. *charts images, navigation buttons, small gifs, navigation bars, logos* etc.)? At least an expiration time and or validator would be necessary. What about Java

Script?

BTW: I don't think that the image handler (who writes the images directly to the servlet output stream) does set any cache-control values that would allow client and/or proxy side caching.

BTW: how does socket-keep-alive work?

Which services (fragments) really need to be encrypted?

- SEPortal (Small Enterprise Portal) services
- telebanking

Server Side Caching: why, what, where and how much?

AEPortal currently uses a caching infrastructure that allows various QOS, e.g. asynchronous requests. This infrastructure should be used for domain object caching needs. It can be found in the package comaepinfrastructure.caching. For caching in other layers the following chapters suggest some other techniques too.

The reason for caching: Throughput

The reason for caching is quite simple: Throughput (and in some cases availability). In some cases – especially when dealing with personalized information - caching will speed up a single new but the effect may not be considered worth the effort – e.g. because the single user case is already fast enough. But on a large-scale site caching will allow us to serve a much larger number of requests concurrently.

This is the reason why in many projects caching gets introduced at a late stage (once the throughput problems are obvious). And it takes some arguing to convince everybody about its importance because it does not speed up a single personalized request as long as there is no *clear distinction between global pieces, individual selections of global pieces and really individual pieces like e.g. a greeting.*

Note: The possibilities for caching are restricted by the application architecture. Caching requires a decomposition of the information space along the dimensions time and personalization

The results from our load-tests are pretty clear: homepage requests are expensive. They allocate a lot of resources and suffer from expensive and unreliable access of external services. And last but not least we would like to avoid DOS attacks caused by simply pressing the re-load button of the browser.

Sometimes caching can also improve availability e.g. if a backend service is temporarily unavailable the system can still use cached data. This depends of course on the quality of the data and excludes things like quotes. The opensymphony oscache module provides a tag library that includes such a feature:

```
<cache:cache
<% try
Inside try block.
<%
// do regular processing here
<%
catch (Exception e)
>
// in case of a problem, use the cached version
```

```
<cache:usecached />
<%
>
</cache:cache>
see Resources, Opensymphony
```

Caching: what

Our original thinking here was that most of our content is NOT cacheable because it is dynamic. A closer inspection of our content revealed that a lot of it would actually be cacheable but this chance has either been neglected or even prohibited by architectural problems.

- Missing assessment of information and content quality and caching possibilities
- Missing separation of server functions from page generation functions

Let’s look at some types of information and their behavior in case of caching: The difficulties for caching algorithms increase from upper left to lower right.

Data / changed by	*Time*	*Personalization*
Country Codes	No (not often, reference data)	No
News	Yes (aging only)	No, but personal selections
Greeting	No	Yes
Message	Yes (slowly aging)	Yes
Stock quotes	Yes (close to real-time)	No, but personal selections
Homepage	Yes (message numbers, quotes)	Yes (greeting etc.)
	Question: how often?	

Country codes are reference data. They rarely change. In AEPortal there is a separate caching mechanism (described below) that deals with reference data only.

All other kinds of data are either changed by time or through personalization and require a different handling. The next best thing to reference data are data that change through time but are at least GLOBAL. Examples are news and quotes which should differ by person (This does not mean that everybody will get the same news)

A greeting (welcome message) does not change at all during a session but is highly personalized. This reduces the impact of caching but does not make it unnecessary for a large site. Reading the same message on every re-load from the DB does not cost a lot but with hundreds of users it is unnecessary overhead.

The homepage is a pretty difficult case. Our initial approach was to not use caching at all because the page was considered highly personalized and also contained near real-time data (quotes)

This was a mistake for the following reasons:

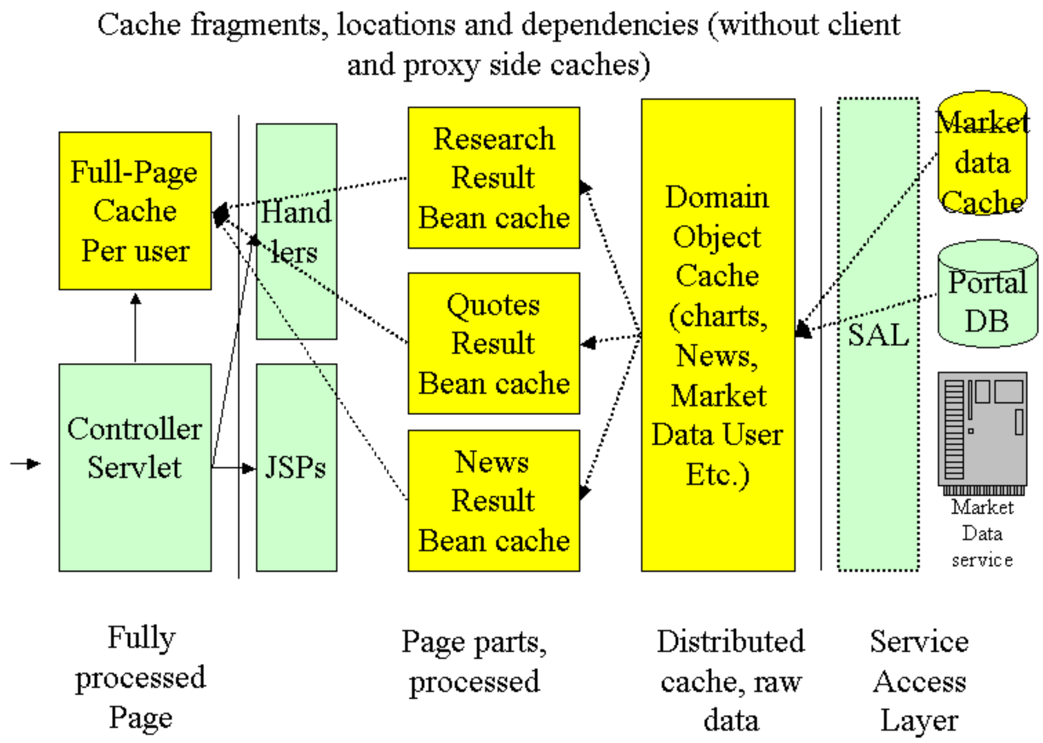
- Page reloads forced by navigation or browser re-size would cause a complete rebuild of the homepage

- According to a report from the yahoo-team (communications of the ACM, topic personalization) 80 % of all users do NOT customize their homepage. This would mean that besides the personal greeting everything else would be standard on the homepage
- Even a very short delay for quotes data would save a lot of roundtrips to the backend service MADIS. Right now we are going for EVERY STANDARD quotes request (i.e. the user did not specify a personal quotes list) to the backend!
- The homepage could be cached in parts too (see below: partial caching)

Caching: Where

Let's first draw a diagram of possible caching locations:

Figure 5.3.



An example for full-page caching is taken again from servlets.com:

Server Caching is Better

The problem with this use of getLastModified() is that the cache lives on the client side, so the performance gain only occurs in the relatively rare case where a client hits Reload repeatedly. What we really want is a server-side cache so that a servlet's output can be saved and sent from cache to different clients as long as the servlet's getLastModified() method says the output hasn't changed.

The existing code for a full page server side cache from Oreilly could easily be extended to support caching of personalized pages. Page descriptions elements should get an additional qualifier to allow this kind of caching.

For a discussion of cache size see below (How Much?)

The full page caching approach suffers from a number of restrictions: While solving the re-load problem (caused by quick navigation or browser re-sizing) it forces us to keep a separate homepage per user. Also, the cache time will depend on the page part with the shortest aging time: we can’t store the homepage for a longer period of time. 30 seconds seems to be the limit. And: if many users do not change their settings or only a few, we keep many duplicates in those homepages.

These problems could be solved by using a partial caching strategy IN ADDITION or as a replacement for the full page cache.

Partial page caching see: http://www.opensymphony.com/oscache/ [http://www.opensymphony.com/oscache/]

Dynamic content must often be executed in some form each request, but sometimes that content doesn't change every request. Caching the whole page does not help because parts of the page change every request. OSCache solves this problem by providing a means to cache sections of JSP pages.

Error Tolerance - If one error occurs somewhere on your dynamic page, chances are the whole page will be returned as an error, even if 95% of the page executed correctly. OSCache solves this problem by allowing you to serve the cached content in the event of an error, and then reporting the error appropriately.

Currently we have a problem providing partial caching: We don’t have the infrastructure to support it properly. Within AEPortal for every request a handler needs to run. This handler allocates resources and creates the result-beans (models). These models are not cacheable (they store references to request etc.). The homepage handler could be tweaked to supply cached model objects without running the respective handlers but this would be a kludge.

If we had this functionality we could assemble homepages from standard parts (not changed by personalization) and personalized parts that cannot be cached at all or a longer time. The standard and non-personalized parts would be updated asynchronously by the cache (using the aging descriptions).

Again, if the 80% rule (yahoo) is correct, this approach would increase throughput enormously. In a first step we would probably cache only the non-personalized parts.

The Domain Object Cache already exists in AEPortal. It is currently used for pictures (charts), profiles and External Data SystemUser: a mixture of personalized and global data. This cache should actually be a distributed one (see below: cloning)

Last but not least the diagram shows a special cache DB for MADIS on the right side. Here we could cache and or automatically replicate frequently used MADIS data (or data from other slow or unreliable external services)

Page Structure, Navigation Design and Performance

Results from Olympic game sites (Nagano etc.) indicate that navigation design has a major impact on site performance. The 98’ Nagano site had a fairly crowded homepage compared to previous sites. This was to avoid useless intermediate page requests and deep navigation paths (see Resources 6).

While the AEPortal homepage is already the place for most of the users interests the page structure could be improved in various ways according to Andy Binggeli. Some of his ideas are:

- Separate the personal parts from default/standard parts. This is especially important for the welcome message. If the welcome message is the only personalized part in an otherwise unchanged homepage we could past the unchanged part easily from a cache.

- Separate the quick database services from slow external access services. Flush every page part as soon as possible

The proposed changes would NOT require us to use frames. But we would have to give up the single large table layout approach and possibly create some horizontally layered tables.

Caching: how much?

A full-page cache that holds every page for every user for a whole session could become very large and pose a performance and stability problem for the Java VM.

Some quick guestimates:

A homepage has an average size of 30 kb. Let’s assume 500 concurrent sessions per VM. Just caching the homepage would cost us 15 Megabyte.

The partial caching of non-personalized homepage parts doesn’t cost a thing. For the personalized parts we would access the domain layer and not the cache.

Domain object cache: This cache holds currently pictures for the charts service (global), profile information (per user) etc. The size of this cache is hard to estimate.

Currently we do not know how big our cache can become on a very busy clone. Critical objects are images and other large entities. How do we prevent the cache from eating up all the memory? Should we store e.g. the charts images in the file system?

Caching: how to?

Domain Object Cache:

Creating a new object cache is simple: A new factory class needs to be created by deriving from the PrefetchCacheFactory and new requester class needs to derive from a request base class.

To allocate a resource from the cache a client either calls PrefetchCache.prefetch() or PrefetchCache.fetch().

Prefetch is intended for requesting the resource asynchronously (i.e. the client does not wait for the resource to be available in the cache). Fetch will – using the clients own thread – go out and get the requested resource synchronously. The client might block in that case.

Both methods will first do a lookup in the cache and check if the resource is already there. And both methods will put a new resource into the cache.

A quality of service interface allows clients to specify e.g. what should happen in case of a null reference being returned from the requester object (should it go into the cache? This could mean that subsequent requests will always retrieve the null reference from the cache instead of creating a new request that might return successfully)

Reference Data caching:

The package comaepinfrastructure.refdata contains the basic infrastructure for reference data handling. A reference data manager (initialized during boot) reads a XML configuration file that describes what data need to be cached and also defines the QOS (aging, reload etc.). A new reference data class can easily be created (probably in comaepAEPortal.refdata package) and a new definition added to RefdataConfigFile.xml

Note: There used to be different RefdataConfigFiles for production, test and development, due to the long load times initially. This has been fixed and there is no longer a real reason for separate configurations. Basically everything is loaded during boot.

We need to clean up the configuration files!

If you need more information on reference data – go and bugger Ralf and Dmitri!

Cache implementations

Reference Data caching

An important part of a caching infrastructure is the quality of service it can provide to different types of data. Some data are only allowed a certain amount of time in the cache. Others should not be cached at all . Some should be pre-loaded, some can use lazy load techniques. Aging can be by relative or absolute time. The currently available QOS for reference data caching are described in the RefData dtd.

Domain object caching

OPEN

Physical Architecture Problems: Clones and Cache Synchronization

The current cache solution has three problems:

- Performance
- Stale copies
- Cache Maintenance

All of them are related to the peculiarities of the current physical architecture, especially the existence of several clones per machine and the lack of session *binding* per clone.

Note: Websphere 3.2.2 provides session *affinity* per clone

In effect this means that a session can use two or more clones on one machine. Since there is a cache per application or clone this in turn means that while one clone might have already cached a certain data – if the next request goes to a different clone on the same machine, its cache again has to load the requested data. Worst case, if we have n clones on a machine we can end up with loading the same data n times onto this machine. This fights the purpose of caching and puts unnecessary loads on network and database.

Besides being a performance issue this raises a much bigger problem: What happens with data that are not read-only? Unavoidable we will end up with the same data having different values in different caches. Currently we can only do two things about it:

- believe IBM that most requests of a session will always go to the same clone (80% likelihood)
- decrease the re-load time (aging interval) of the cached objects (This is not even an absolute aging yet)

Note: the 80% have not been tested yet!

User access tokens and profile entries are the most likely candidates to cause problems here.

Cache maintenance is impossible too because we have no way to contact the individual clones. If we could we could as well synchronize the caches in case of changes....

This has already hit us once: In case of a minor database change which requires the database to be shut down and restarted there is a chance that id’s have changed. Without recycling the clones they will have still the old values in the cache.

If we give up the idea of session affinity to ONE node – e.g. if we want to achieve a higher level of fail-over, then we have the same problem between ALL nodes!

Inter-clone communication is a very important topic for the re-design. Websphere needs to provide a mechanism here or cloning does not make sense in the longer run.

Solution One: A messaging system

In this case a change to the database would be sprayed to all clones – possibly using a topic based publish/subscribe system and the clones would then update the data.

The Domain object cache on each clone or application instance would then need to subscribe for each cache entry to get notification of changes.

On top of solving the cache synchronization problem this would also give us a means to inform running application instances about all kinds of changes (configuration changes, new software, database updates etc.)

Solution Two: CARP

CARP (Cache Array Routing Protocol) could be used to connect the individual caches of all clones.

"CARP is a hashing mechanism which allows a cache to be added or removed from a cache array without relocating more than a single cache’s share of objects. [..] CARP calculates a hash not only for the keys referencing objects (e.g. URL’s) but also for the address of each cache. It then combines key hash values with each address hash value using bitwise XOR (exclusive OR). The primary owner for an object is the one resulting in the highest combined hash score." (15)

This solution would only provide a means to synchronize (actually, to avoid the synchronization problem) several caches.

Portlets

From Handlers to Portlets

So far we have learned the following deficiencies of our model 2 architecture:

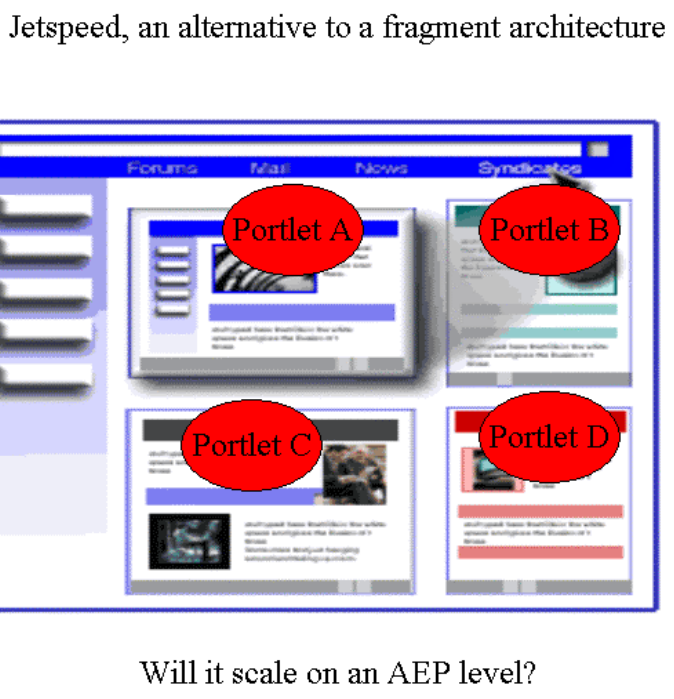
- The concept of "services" is vague. Sometimes it means bits of the screen and sometimes it means an application or backend data source.
- "Handlers" are a procedural concept and do not allow clever caching because the abstractions for the things that would need caching do not exist (fragments)

"Portlets" were designed to better represent the aggregation of information within a portal page. The next chapter shows that "Portlets" are still somehow vague – are they applications or information? What kind of user experience and degree of information integration do they provide? What are the essential differences to our handler based approach? And finally: are portlets enough?

Portlet: Application or Information?

At first glance portlets seem to represent pretty much the same "boxed" portal page concept as our handlers above. An independent piece of screen real-estate is represented by an individual portlet.

Figure 5.4.


Jetspeed, an alternative to a fragment architecture

Will it scale on an AEP level?

The following text from IBM gives (a lot) more definitions. I have highlighted the most important bits: http://www-4.ibm.com/software/webservers/portal/portlet.html [http://www-4.ibm.com/software/webservers/portal/portlet.html].

What is a Portlet?

"Portlets are the visible active components end users see within their portal pages. Similar to a *window* in a PC desktop, each portlet owns a portion of the browser or PDA screen where it displays results. Portlets can be as simple as your email or as complex as a sales forecast from a CRM application.

From a user's view, a portlet is *a content channel or application* to which a user subscribes, adds to their personal portal page, and configures to show personalized content.

From a content provider's view, a portlet is a means to make available their *content*

From a portal administrator's view, a portlet is a *content container* that can be registered with the portal, so that users may subscribe to it.

From a portal's point of view, a portlet is a *component* rendered into one of its pages.

From a technical point of view, a portlet is *a piece of code* that runs on a portal server and provides content to be embedded into portal pages. In the simplest terms, a portlet is a Java servlet that operates inside a portal.

Portlets are *often small portions of applications*. However, portlets do not replace an application's numerous displays and transactions. Instead, a portlet is often used for occasional access to application data or for high profile information that needs to be *displayed next to crucial information from other applications*. In some cases, this is like an executive information system or a balanced scorecard key performance indicators display. In other cases, it may simply be *a productivity enhancement where a user can have all the tools and information needed to quickly access multiple applications, documents, and results*. For example, a procurement analyst may want to see online vendor product catalogs with prices side by side with current inventory levels from the ERP system, and this next to a business intelligence analysis of item usage for the last 18 months. In any case, a portlet provides a real time display of vital information based on the users preferences.

Portlets can be built by the Information technology department, systems integrators, independent software vendors, and of course, IBM. Once a portlet is developed and tested, it is stored into the portlet catalog. By browsing the portlet catalog, an end user can then select a portlet and place it into one of their own portal pages."

To summarize, a portlet can be:

- a piece of code
- a container
- a piece of content
- an application
- a window

The definition mixes the results of processing (content) with the means of processing (code, application, container). The difference is quite important: I cannot cache the code but I can cache results!

The main differences to the handler approach:

- A portal using portlets includes the infrastructure to dynamically add and delete portlets. In that sense portlets have a better "handle" for maintenance.
- Portlets typically do their own rendering. The main window simply glues these bits and pieces together.

Portlets seem to be the proper mechanism to create "Windows" or yahoo style portal pages combining independent pieces of information or applications for convenience reasons.

They do NOT provide an infrastructure for sites that want to provide a much higher information integration level – in the sense of combining the information form various sources (applications, data sources etc.) into an integrated information of higher value.

The portlet approach reflects the different information sources behind the enterprise portal, it does NOT integrate them in the sense of integrated and linked CONTENT.

High performance sites would need a separate fragment caching layer for easy and quick assembly of results without going through portlet processing. This layer needs to provide naming and addressing of result fragments.

Fragments

Fragments are a concept independent of base technology assumptions like J2EE or Apaches Jetspeed etc. Fragments provide an information view on content and stay valid even if base technology changes.

Fragment Definitions

Fragments are pieces of information that have an independent meaning and identity in the user's conceptual model. This can be a piece of news or research or a single quote.

Fragments can contain other fragments or references to those.

Fragments have names and identities and can be associated (via a catalog) with a system identifier that allows the system to load or store a fragment through a specific service in a specific place.

Fragments can have one or more subjects associated. They can form a dependency chain of fragments. If fragments lower in the chain change, the higher fragments need to be re-validated or updated.

Fragments are the basic unit for caching.

Fragments can be persisted and have a read/write interface.

Fragment Chaining

Fragments appear in various formats. An example:

The subject "news service" of the homepage includes several fragments in one fragment chain.

- The database rows for all the news articles in the system.
- The Domain Object that may have already filtered some content from the DB
- An XML version of the same content
- A personalized (filtered, selected) XML version for user A
- A rendered version of the personalized XML version for user A

Html XML(pers) XML(common) Domain Object Database Row(s)

The difference between these fragments is that various transformation processes have been applied. A high-speed site needs to store fragments in various formats to avoid repeated and costly transformations. At the same time the site needs to guarantee the consistency of the subject e.g. the news block in the homepage by invalidating all fragments in the chain.

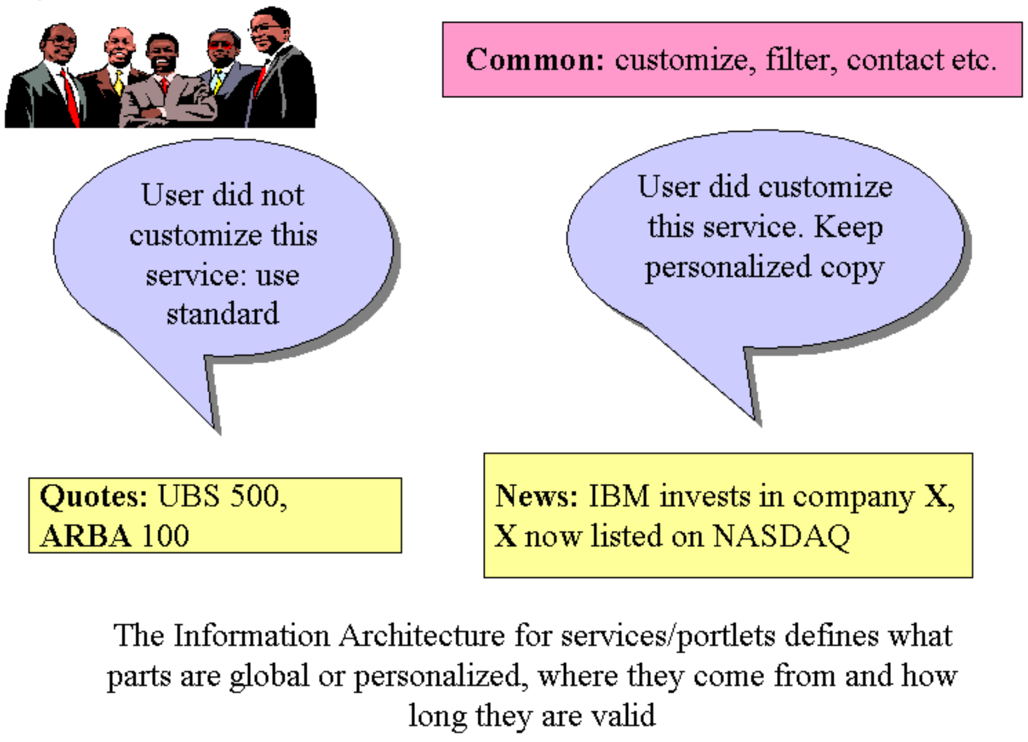
Composite Fragments

A single page as well as a homepage can be composite fragments. Composite Fragments are described by Fragment Definition Sets FDG (which are fragments as well that define what can/must go into a certain fragment. The composite fragment contains complete sub-fragments or references to other fragments).

The fragment definition sets form an object dependency graph (ODG). This graph is used to invalidate fragments and fragment chains. The real invalidation needs to be performed by going through the object dependency graph formed by the fragment instances themselves.

The lifetime of individual fragments can be very different and is defined either by the FDS or other rules.

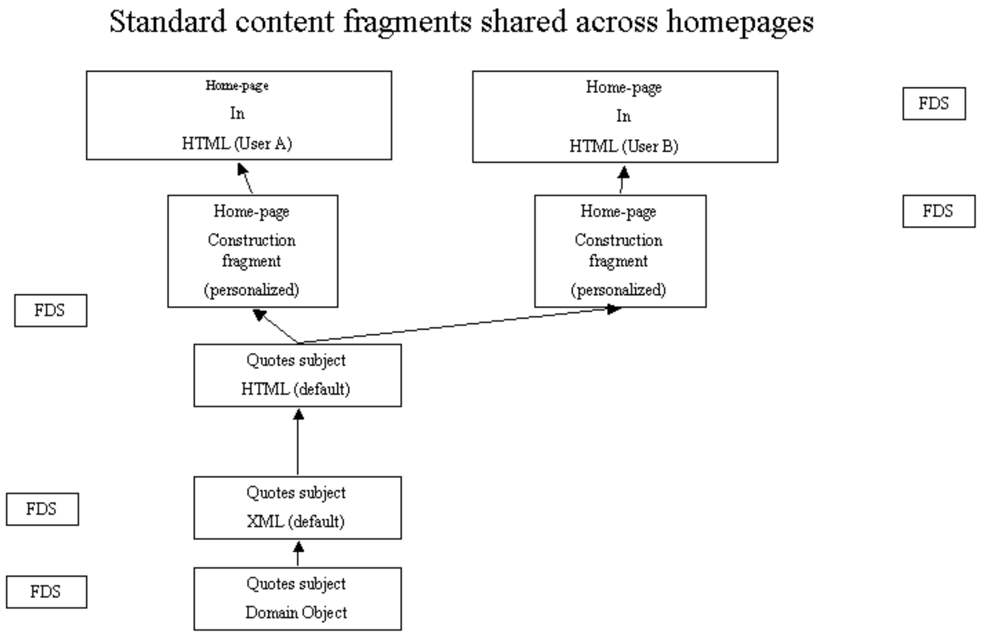
Figure 5.5.


Common: customize, filter, contact etc.
User did not customize this service: use standard
User did customize this service. Keep personalized copy
Quotes: UBS 500, ARBA 100
News: IBM invests in company X, X now listed on NASDAQ
The Information Architecture for services/portlets defines what parts are global or personalized, where they come from and how long they are valid

Fragment Sharing

Every Fragment can be referenced from various other fragments. As long as a fragment is only referenced, an update of this fragment will immediately become effective. Derived fragments still have to be updated too if they somehow embed the sub-fragments content.

Figure 5.6.


Standard content fragments shared across homepages
Home-page In HTML (User A)
Home-page In HTML (User B)
FDS
FDS
Home-page Construction fragment (personalized)
Home-page Construction fragment (personalized)
FDS
Quotes subject HTML (default)
FDS
Quotes subject XML (default)
FDS
Quotes subject Domain Object

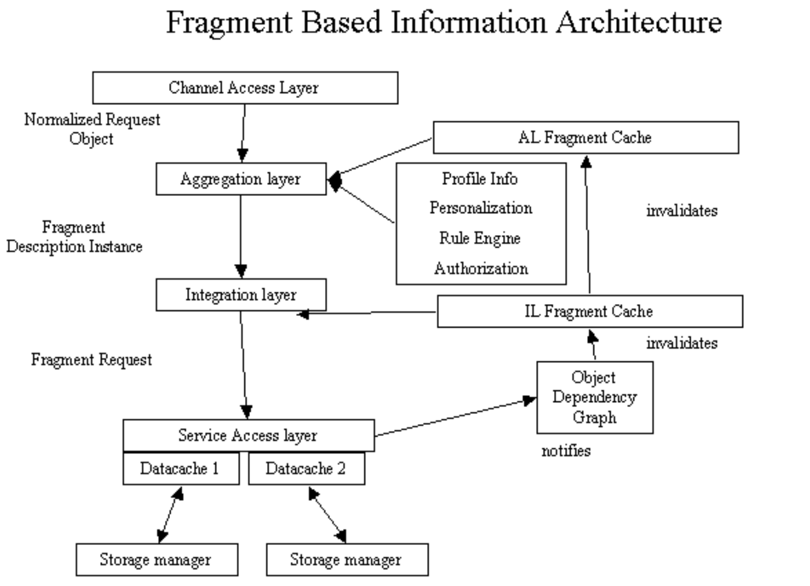
Fragment Validation

Every Fragment has an associated Validator FV. This object will be contacted if a system component needs to find out if a certain fragment is still valid. If the answer is no, the fragment itself should be invalidated. In addition to this The fragment should know how to update itself – or at least contain all the meta-information or configuration information to make this possible.

Fragment Architecture Overview

The diagram below gives an overview of the information flow in a portal. The explanation starts as usual with an incoming request but there are certainly asynchronous information gathering processes active at the same time (read-ahead etc.)

Figure 5.7.



Goal: minimize backend access through fragment assembly (extension of IBM Watson research)

1. All incoming requests from various channel go through the channel access layer (CAL). This layer does a normalization of the requests by creating a platform independent request/response object pair.

2. The request object enters the aggregation layer (AL). It is the responsibility of this layer to map the request to a certain fragment (a page or parts of it).

3. The first responsibility of the AL layer is to create a validator for the requested fragment. Using this validator the layer can question cached information (e.g. a full-page rendered information cache) whether the fragment is cached and still valid. If the fragment is still valid, AL layer returns immediately.

Alternative: We could use Data Update Propagation (DUP) to notify caches about invalidation and avoid the validator concept at this level. Validators would then only be necessary if something changes and the scope of associated changes needs to be determined.

1. In case no suitable fragment was cached, a fragment definition instance for this user is then either created (contact the profile information) or retrieved from a cache (with write through) The fragment definition instance contains the fragment type information, filtered by the personal settings and authorization information:

User X requests fragment "homepage". The type definition for the fragment homepage defines that an instance of a homepage fragment can contain fragments of type "quotes" and "news". The access control subsystem and the user profile contains the information that both fragments are active (not minimized) and the "news" fragment has been personalized (topics, rows) while the quotes fragment is the default quotes information (no personal selections, no GUI changes like more rows etc.)

Please note: if the fragment definition instance has been cached too, then the aggregation layer saves the work to create a user specific fragment request!

1. The fragment definition instance is now forwarded to the integration layer (IL). The integration layer checks its caches for fragments needed to fulfill the request. The Ids of the fragments encode whether they denote personalized or common versions.

Best-case scenario is of course if the fragments all exist in the cache and are valid. (Do they even exist if they are not valid?)

1. If a fragment does not exist in the IL cache, the fragment request is forwarded to the service access layer (SAL) to retrieve the fragment. SAL will itself cache the raw domain data, the fragment ends up in the IL fragment cache.

Please note: system information from catalogs map fragments to services. The fragment itself does not contain that information. Aggregation and Integration only deal with fragments, not with services (that will probably use something like a handler to retrieve fragments)

Pooling

AEPortal uses a pooling infrastructure that allows various QOS, e.g. aging by time or request count. This infrastructure should be used for all pooling needs.

Note: Pooling is not the same as caching because cached objects need a unique name or id so that clients can address them. This is not necessary for pooling. Still, both could probably be implemented using the same base classes. This would be a topic for the re-design.

Implementation

AEPortal uses a generic object pool called "Minerva". Minerva is now part of the jboss open source project (http://javatree.web.cern.ch/javatree) and does also advanced JDBC connection pooling (JDBC SE, 2 phase commit support) but we only use the generic object pool. It is in the package org.jboss.Minerva.pools. Infrastructure now has a package comaepinfrastructure.pooling which includes standard factories and a main pool factory. This package assumes right now a couple of default pools, e.g. DOMParserPool but new pools can be created programmatically. The process is similar to the creation of new caches: You need a new factory with a method to create a specific object and – optionally – have that object implement the PooledObject interface.

What is missing:

- a xml description of default pools so that the system can configure the pool factory at boot time (like we do with reference data)
- a way to do the same at runtime from within applications or services (to make AEPortal dynamically extensible)
- more quality of service features: aging by time and request.

Documentation: twiki, search for "ObjectPool".

Pooling: why, what and how much?

We pool things because we want to save either time – if it takes a lot of time to build an object – or memory – if an object has a large footprint.

The "what" part is harder to answer: Typically heavyweight objects that are not re-entrant because they keep some state e.g. XML Parsers. Right now it is necessary to give two threads two XML Parser instances because they might each install a different entity manager etc. There is only one requirement: the pooled objects need to be "resettable", i.e. a new client thread (or the pool itself when the object is returned) can return the object to its initial state.

Typically objects that get pooled are:

- threads
- database connections
- XML Parsers
- Network connections

Pooling: how?

Unlike caching pooling follows an allocate/free pattern. Objects that are not "freed" after use are no longer available for other threads.

Sometimes it is hard to retrofit a piece of code with pooling because the moment where the pooled resource needs to be returned to the pool cannot be determined.

Example 5.2. A pooling example:

```
// before pooling
Class ResourceUser {
Public ResourceUser () {
Resource mResource = new Resource(); // get a new instance
mResource.setSomeMode(x); // initialize it
}
Public useResource() {
String result = mResource.parse(foo);
}
}
Given this class, when would you return a pooled resource to the pool?


// with pooling
Class ResourceUser {
Public ResouceUser () {
}
private Resource getResource() {
Resource mResouce = (Resource) pool.getResource(); // get a pooled instance
mResource.setSomeMode(x); // initialize it
}
private void freeResource(Resource res) {
pool.releaseObject(res);
}
Public useResource() {
Resource res = getResource(); // get resource
String result = res.parse(foo); // use it
FreeResource(res); // return it immediately
}
}
```

Note

You can no longer let objects of class PoolUser be collected by the Garbage Collector without returning the pooled resource beforehand. The code is designed for short time usage of a resource. In the non-pooled version the resource is private to class PoolUser. This is expensive but the good side of it is that this cannot create a bottleneck. In the pooled version – if class PoolUser does not call freeResource() after every use – we have created a bottleneck in the system. Of course, if class PoolUser would use the resource in a tight and non-blocking loop, it would probably hold on to the one and avoid the pool management overhead.

GUI design for speed

GUI design has both an subjective and objective impact on the speed of a portal. A subjective result of GUI design is e.g. how the user experiences the load times for the homepage.

Incremental page loading

As we have seen the homepage processing is fairly intensive and will take a while. This leads to a wait time of up to 20 seconds for a user - a number that is usually not acceptable according to the usability literature (which talks about the magic 8 seconds after which a user leaves the page). While this may not be true in case of enterprise portal - because users save a lot of time due to high service and information integration (Single-Sign-On, aggregated reports etc.) the GUI design can nevertheless reduce the waiting time subjectively. The trick can be seen e.g. at sites like www.excite.com and works like that:

- After 2-4 seconds excite shows a small page header containing static information.
- After 7-8 seconds excite shows a middle section showing dynamic information.
- After 16-19 seconds excite shows dynamic and personalized information and completes the page.

The effect on the user experience is quite drastic: compared to a homepage that takes only 14 seconds to display but does *show nothing while loading* the incremental load *feels much faster* even so it is objectively longer. Go and try that - it really works.

Information ordering

To use this effect the GUI design can also have an objective impact on load time by ordering the content in a way optimized for incremental load.

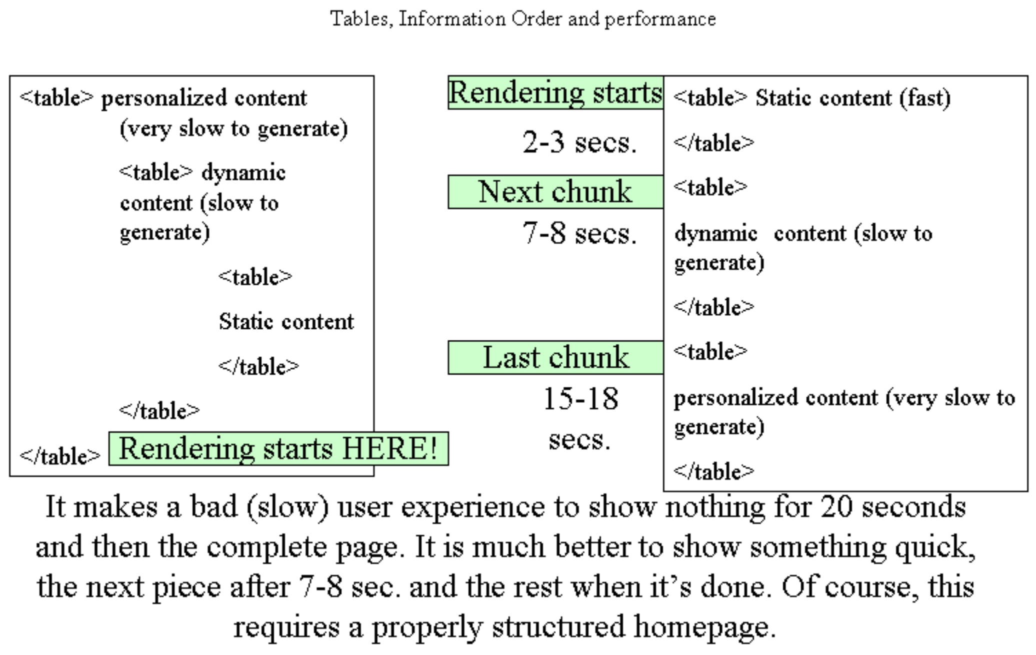
Gui Information ordering

In most cases this means to design the GUI to display static content on top of the page, then to put dynamic but non-personalized content in the middle and reserve the bottom space (the "longest" area) for dynamic and personalized information which takes presumably the longest to collect on the server side.

The big table problem

What prevents incremental loading technically is the use of one big table that embraces all the homepage content. While this is nice from a layout point of view it prevents most browsers from rendering those parts of the homepage that have been received already. Instead - the browser waits for the closing table element (which is the end of the homepage html stream and can be around 50Kb later) to start rendering.

Figure 5.8.


Tables, Information Order and performance
<table> personalized content (very slow to generate)
<table> dynamic content (slow to generate)
<table>
Static content
</table>
</table>
</table>
Rendering starts HERE!
Rendering starts
2-3 secs.
Next chunk
7-8 secs.
Last chunk
15-18 secs.
<table> Static content (fast)
</table>
<table>
dynamic content (slow to generate)
</table>
<table>
personalized content (very slow to generate)
</table>
It makes a bad (slow) user experience to show nothing for 20 seconds and then the complete page. It is much better to show something quick, the next piece after 7-8 sec. and the rest when it's done. Of course, this requires a properly structured homepage.

Throughput

Much of this document has already been dealing with performance and throughput issues. Here I would like to collect some more ideas from the AEPortal team on throughput or performance improvements

Java Pitfalls

Certain Java styles will have a very strong negative impact on system throughput:

- excessive object creation
- excessive Garbage Collection (caused by a lot of heap activity)
- thread context switching (especially with early JDK releases)
- exception throwing

- excessive synchronization

In the end it was necessary to walk through Jack Shirazis book on Java performance to fix the worst problems. It turned out that e.g. throwing and catching an exception is worth a couple of hundred lines of Java code or up to 400 ms.

DO NOT USE EXCEPTIONS IN PERFORMANCE CRITICAL SECTIONS – EVEN IF IT IS YOUR "STYLE"!

Excessive object creation and garbage collection can only be avoided using advanced caching and pooling strategies and last but not least interface designs which avoid useless copy's – this goes deep into architecture.

The old saying: first get it going and optimize afterwards DOES NOT WORK HERE!

The latest garbage collectors work generational: they run through older data less frequently. This is very good for large caches which would otherwise cause a lot of GC activity.

Excessive synchronization is very common. To prevent the framework classes in the critical performance path are either singletons or have static methods.

Double-checked locking as a means to prevent multiple copies of a singleton and at the same time avoiding the performance penalty of using "synchronized" DOES NOT WORK. There seems to be NO workaround right now besides falling back to making the factory method synchronized or "eager initialization" (28).

The following is *WRONG*:

Example 5.3. Broken multithreaded version of "Double-Checked Locking" idiom

```
class Foo {
private Helper helper = null;
public Helper getHelper() {
if (helper == null)
 synchronized(this) {
if (helper == null)
helper = new Helper();
}
return helper;
}
// other functions and members...
}
```

XML processing

The parsing and or writing of XML documents or requests turned out to be quite expensive. The recently created weakness-analysis already suggested to replace the DOMParser with a SAXParser. While this would probably improve the performance and throughput, an even more powerful idea was suggested: generate specialized parsers for certain DTDs. This technique has been successfully applied in other projects [CMT99]. The specialized parsers are able to process the XML requests at nearly I/O speed (e.g. like the Expat parser by J.Clark) with minimum memory footprint.

Note:

When considering performance and scalability, the first concern with the DOM approach is the effect it has on system resources. Since DOM parsers load the entire XML document into memory, the implementation must have the physical memory available for this task. This requires the application to manage overflows as there's no real recourse for a document that's too large. Perhaps more important is how this limitation impacts the support for the number of documents that can be opened in parallel by the calling application. Since the DOM specification doesn't enable an implementation to process the document in sections, this request can add significant overhead to the processing of multiple documents. Furthermore, current parser implementations are not reentrant; that is, they can't allow multiple data sets to be mapped against a single DOM held in memory. The upfront resource costs of using the DOM approach are more than justified if the core function of the application is to substantially or repeatedly modify the content or the structure of the document. In this case working with the DOM allows efficient and reusable application calls that can interface directly with the XML document. Procedurally, if the calling application requires this level of access to the entire XML document or the ability to process different sections of the document, the use of the DOM API may be warranted.

SAX Approach

While a DOM parser reads the entire document into a tree structure before allowing any processing, an event-based parser allows the application to have a conversation with the XML document by communicating events in response to invoked methods by the application's implemented handler. Because of this conversational approach, the memory and initial processing requirements are lower for an application processing a document using SAX. This resource efficiency, though, requires the application to manage the various conversation handlers that are necessary to fully communicate with an XML document. Managing these handlers becomes the responsibility of the calling application. (from the XML Journal, http://www.sys-con.com/xml/archives/0203/patel/index.html)

A related problem is the mapping from XML elements to java classes (e.g. in External Data System). Class.forName("TagName") is VERY expensive *within servlets*. It uses the *servlet* classloader which is very costly.

This mechanism is used e.g. in MarketDataSelectorImpl, NewsDataSelectorImpl and ResultTagsImpl.

The same mechanism (but not for XML processing) is used extensively by the ProfileManager – another reason to get rid of this component.

XML-RPC type communication performance

Communication with External Data System uses XML messages while e.g. Telebanking uses the CORBA interface of External Data System.

The use of e.g. SOAP as a communication protocol underlying an rpc mechanism carries a performance degradation factor of 10 compared to Java RMI. See "Requirements for and Evaluation of RMI Protocols for Scientific Computing".

XML-RPC type communication is surprisingly fast for short messages. This shows that serialization/de-serialization is expensive, especially if an XML to object mapping is performed using Java reflection. Specialized "PullParsers" could improve the performance quite a bit as well as avoiding to convert too many XML elements into Java objects.

Does it pay to use a special parser for XML-rpc? From Graham Glass:

When my company decided to create a high performance SOAP engine, we started by examining the existing XML parsers to see which would best suit our needs. To our surprise, we found that the commercially available XML parsers were too slow to allow SOAP to perform as a practical replacement for technologies like CORBA and RMI. For example, parsing the SOAP message in Listing 1 took one popular XML parser about 2.7 milliseconds.

Example 5.4. soap example

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/1999/XMLSchema">
 <SOAP-ENV:Body>
 <ns1:getRate xmlns:ns1="urn:demo1:exchange"

 SOAPENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 <country1
 xsi:type="xsd:string">USA</country1>
 <country2
 xsi:type="xsd:string">japan</country2>
 </ns1:getRate>
 </SOAP-ENV:Body>
 </SOAP-ENV:Envelope><SOAP-ENV:Envelope

 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/1999/XMLSchema">
 <SOAP-ENV:Body>
 <ns1:getRate xmlns:ns1="urn:demo1:exchange"

 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 <country1
 xsi:type="xsd:string">USA</country1>
 <country2
 xsi:type="xsd:string">japan</country2>
 </ns1:getRate>
 </SOAP-ENV:Body>
 </SOAP-ENV:Envelope>
```

Since a round-trip RPC message requires both the request and the response to be parsed, using this popular parser would mean that a SOAP RPC call would generally take at least 5 milliseconds to process, and that doesn't include the network time or the processing time on either side. This doesn't sound too bad until you consider that a complete round-trip RPC message using RMI takes about 1 millisecond. So before giving up on ever building a SOAP engine that could compete against traditional technologies, we decided to experiment with building our own XML parser. (See http://www.themindelectric.com/products/download/download.html for how to download Electric XML, which developers may use without charge for most commercial and noncommercial uses.) XML compression is also discussed in the context of XML-RPC: According to Gerd Mueller (gerd@smb-tec.com) the ozone/XML project uses "binary XML" to transfer XML from the clients to the database server and back through a socket connection. It is based on some work of Stefano Mazzocchi from Apache/Cocoon.

He called it 'XML compiler' and it compiles/compresses SAX events.

Database performance

Our database is currently NO bottleneck – I wish it were!

Why is our database no bottleneck? Simply because we spend less than a second in our database driven services combined and several seconds in our services that access external sources like MADIS.

But once this problem is fixed THE DATABASE WILL BE OUR BIGGEST BOTTLENECK- confirmed by just about every paper on dynamic web content delivery (see Resources below).

And the reason for this is not bad DB design or large amounts of data transferred in single requests. It is simply the huge number of requests per personalized homepage or regular page driven against the DB.

I had a hard time to get everybody to recognize this. Especially if the amount of data in retrieved was considered to be small. Or the DB was used for filtering. This is true for regular stand-alone web applications. It does NOT apply for an enterprise portal that runs a lot of services in parallel just to satisfy ONE client request!

The golden rules:

- If it don’t change – cache it!
- If it changes only every once in a while – use the cache automatic reload feature to retrieve it
- Don’t "abuse" the DB to get a convenient filtering mechanism – even so you could and should do so in a regular application.
- If it’s personalized – still do cache it! The client might do re-loads.
- It it’s small and personalized – put it in session state (e.g. greeting)

Connection Hold Time: given the fact that the number of connections to a DB is limited for each application server it is vital for the overall performance of the portal that requests do not hold on to a DB connection for longer periods of time.

The calls to getConnection() and freeConnection() have been instrumented to record the hold time. The architectural problem lies in the fact that proper system throughput depends on proper behaviour by the service developers and cannot be enforced by the system itself. Watch out for coding patterns that allocate the DB connection and then let it flow through nested sub-routine calls. This will typically result in overly long hold times.

The logging mechanism could be used to print warnings if a certain time limit is exceeded!

Paging

Currently we do not have framework support for selective and partial DB reads – e.g. caused by a user paging through documents. JDBC2.0 provides support for this. Some of it has been backported to JDBC1.0.

This is a hot topic within the servlet discussion groups.

Http Compression

Several packages are available that provide on the fly compression of http content (29, 17). Reductions in size of more than 90% are possible, reducing the download of e.g. the 50kb homepage substantially.

The basic requirement is that the browsers follow IETF (Internet Engineering Task Force) Content-Encoding standards by putting

"Accept-Encoding: gzip, compress"

in the http header.

Ideally compression should not happen at the application server. Reverse proxies seem to be a good place to perform compression without putting further load on the application server(s).

SSL Sessions and Acceleration

SSL processing puts a heavy load on systems. Not all systems are well equipped to do key processing at high speed (e.g. PCs outperform Sun Ultras by quite a margin).

Some numbers:

Depending on how many server requests are necessary to display a page the rendering time can grow from 5 seconds (http) to 40 seconds or more (https) (33).

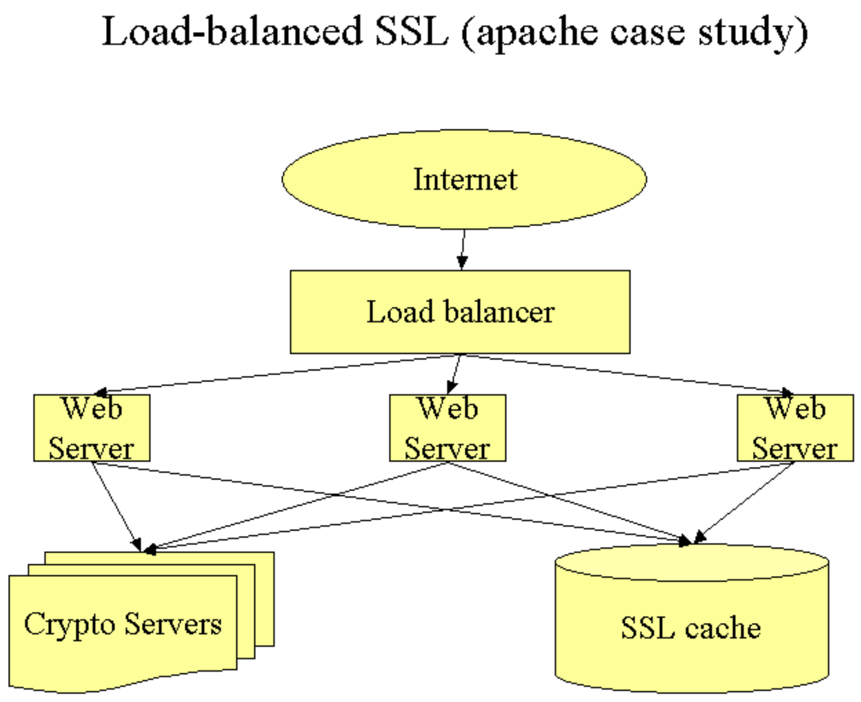
A standard Web server can handle only 1% to 10% of its normal load when processing secure SSL sessions (34).

Note: KNOW YOUR PAGES! It is absolutely vital to know the structure of your pages and what it means in http-terms, e.g. how many individual requests are necessary to build a page.

Note: KNOW YOUR SSL PERFORMANCE! It is absolutely vital to have proper performance data on the throughput of your SSL processors (reverse proxies etc.). If the physical architecture seems to always get bigger and bigger here you can assume that the architecture is based on guesswork.

The authors of (30) suggest the following architecture:

Figure 5.9.



This architecture allows the sharing of SSL sessions e.g. for concurrent requests during embedded image load or generally to achieve a better load-balancing. Without a shared SSL cache (e.g. a shared memory session cache only) moving to a different web server would cause new SSL negotiations.

Note: Currently there are no performance data available. The portal physical architecture now includes SUN E4500 with 8 GB Ram and 6 CPU's to perform authentication and SSL management. Also missing is the separation of static images from dynamically served content. Do the concurrent requests for embedded images have to go through SSL or not?

Reader/Writer Locks

Using synchronized blocks for access to independently updated resources is very costly. The solution is to use Reader/Writer locks.

From Billy Newport (32):

"The reader writer locks have nothing to do with reading and writing actually. It is really a lock that splits parties that need access to the resource in to two groups. Anyone from the first group can concurrently access the resource. However, when someone from the second groups needs access, we block everyone in the first group until we get access. So, the second group has priority over the first group."

Http session handling in the Application Server

(37) suggests several ways to improve session performance:

- use the session cache of the application server
- optionally use manual session persistence. Architectural support for this has been built into the portal already by protecting "UserState" with a dirty flag.
- Turn off automatic session creation from JSPs (if not needed or multi-framed JSPs are used)
- Run the IBMTrackerDebug servlet to test performance

Caveats:

Session sharing between servlets has consistency, classpath and performance problems (no multi-row capability, servlets do not have class files for objects from other servlets, servlets need to deserialize many objects they don't use. A multi-framed JSP cannot combine 2 Web Applications because they cannot read in the same session concurrently.??? Different web-applications used in one JSP break session affinity. When application server security is enabled, all resources need to be either secured or unsecured, no mix and match is possible.

Asynchronous Processing

The amount of processing that can happen during a time of a homepage request is limited. Processing is much shorter if external requests can be avoided e.g. by having a daemon extract data asynchronously.

Examples of asynchronous processing:

- background replication between External Data System and AEPortal caching, driven by user profile information
- splitting the homepage into several multi-pages, some of them are pre-loaded in the background
- starting an asynchronous requester during first access and – driven by profile values – fill in the domain object cache.
- Use a Java Messaging System to request and publish data within the application and between clones.

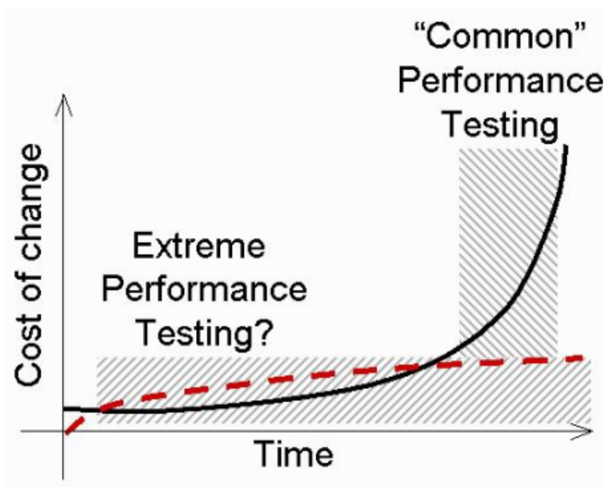
Extreme Load Testing

Most enterprise portals lack performance and or stability during the first couple of releases (e.g. Deutsche Bank, yellowworld etc.) and AEPortal was no exception. Current project methodology was not helpful here because it did not recognize the different needs of an enterprise portal:

- There is NO proven knowledge about how to build a large-scale enterprise portal
- Most of the current technologies are IMMATURE (Java, Web Application servers etc.) bug ridden or have low performance and stability (Java1.1 on multiprocessors etc.)
- Load-tests are no longer simple acceptance tests (spend a couple of weeks and put the checkmark behind the milestone). The load-tests are needed to MAKE the portal perform – in other words: they will cause a cycle of engineering and software changes over weeks and months. There is no chance of an enterprise portal to support an open user group right from the beginning.

Figure 5.10.

Portal Load Tests are “Extreme”



Source: Ted Osborne from Empirix

- The load tests do not only cause software changes. In many cases they force the enterprise to re-engineer some of its backend services because they do not scale in the context of the portal. Project management needs to manage and track this process.
- The end-to-end environment is extremely complex and CANNOT be tested in one go with a huge enterprise portal application. It is absolutely nonsense to start portal testing BEFORE every component in the whole processing chain (from load-balancing and reverse proxies web server, web application server(s), databases and backend service connections) has been tested INDIVIDUALLY.

Chapter 6. Portal Architecture

Domain Analysis

Figure 6.1.

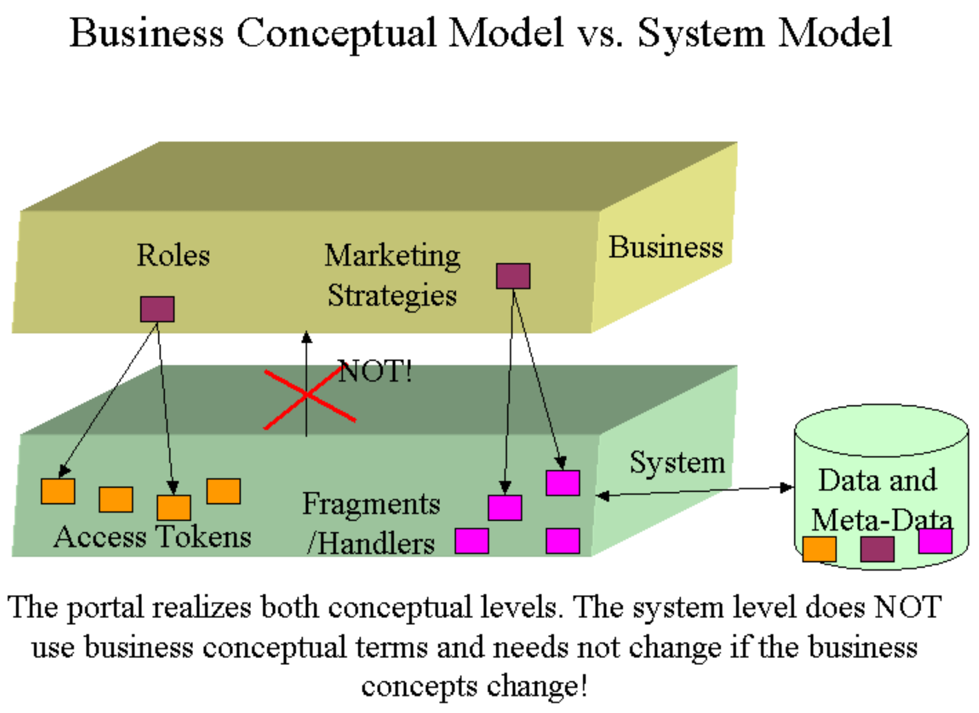
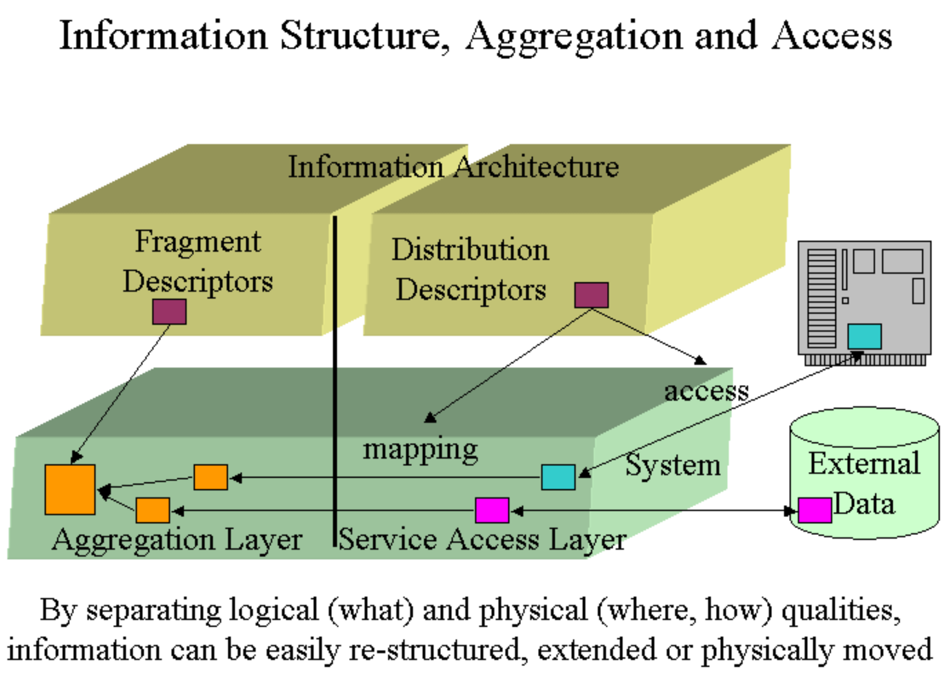
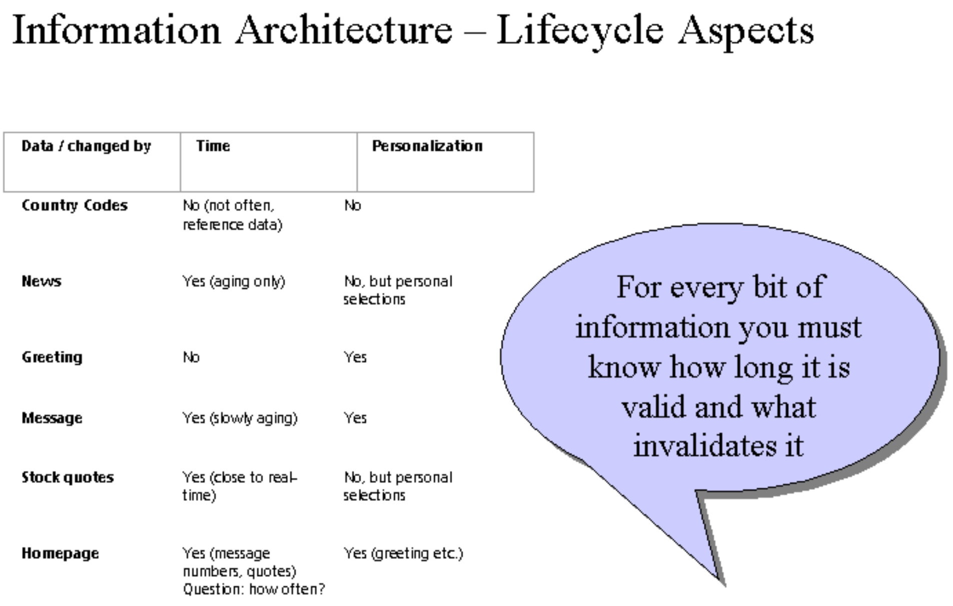

Business Conceptual Model vs. System Model
Roles
Marketing Strategies
Business
NOT!
System
Data and Meta-Data
Fragments /Handlers
Access Tokens
The portal realizes both conceptual levels. The system level does NOT use business conceptual terms and needs not change if the business concepts change!

Figure 6.2.


Information Structure, Aggregation and Access
Information Architecture
Fragment Descriptors
Distribution Descriptors
access
mapping
System
External Data
Aggregation Layer
Service Access Layer
By separating logical (what) and physical (where, how) qualities, information can be easily re-structured, extended or physically moved

Information Architecture

Figure 6.3.



Information Architecture – Lifecycle Aspects

Data / changed by	Time	Personalization
Country Codes	No (not often, reference data)	No
News	Yes (aging only)	No, but personal selections
Greeting	No	Yes
Message	Yes (slowly aging)	Yes
Stock quotes	Yes (close to real-time)	No, but personal selections
Homepage	Yes (message numbers, quotes) Question: how often?	Yes (greeting etc.)

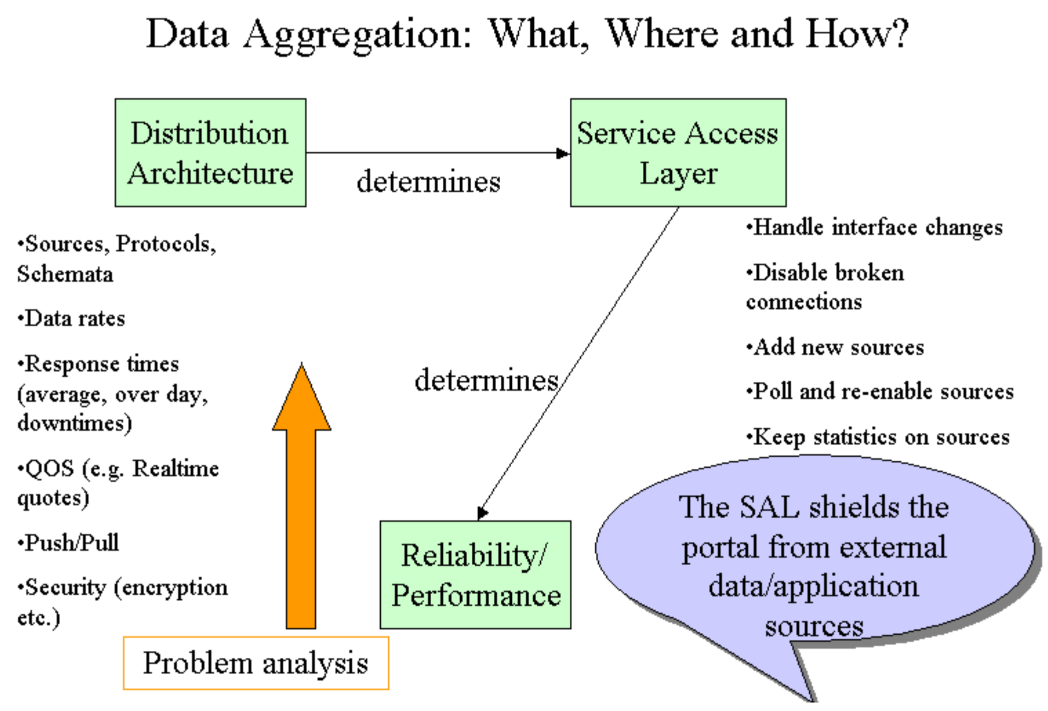
For every bit of information you must know how long it is valid and what invalidates it

Distribution Architecture

Service Access Layer

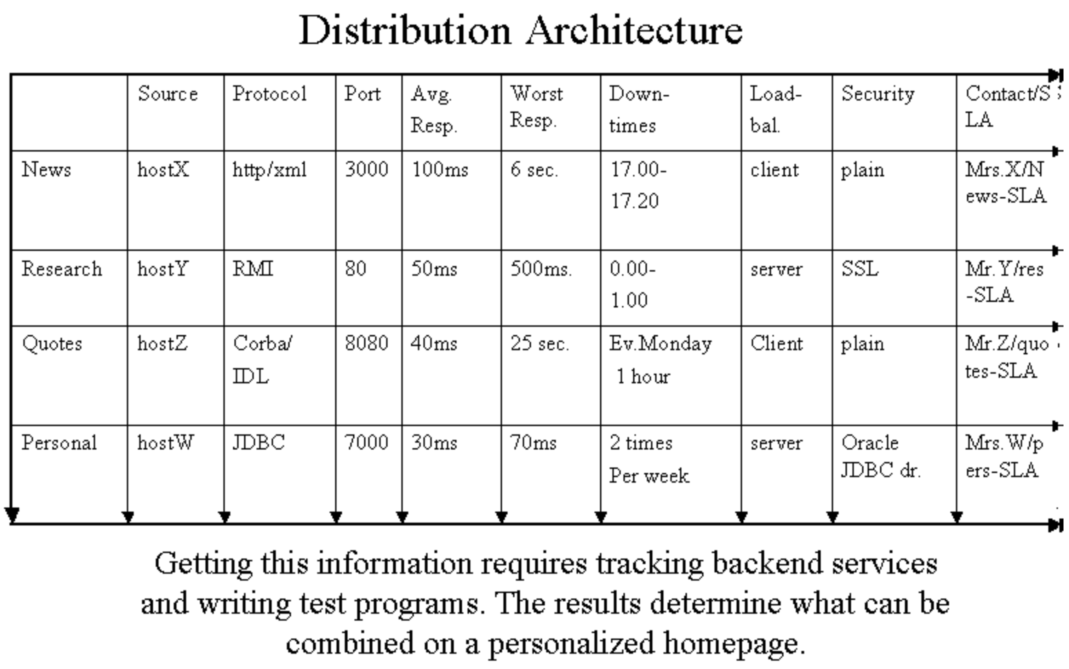
Data Aggregation

Figure 6.4.


Data Aggregation: What, Where and How?
Distribution Architecture
determines
Service Access Layer
•Sources, Protocols, Schemata
•Data rates
•Response times (average, over day, downtimes)
•QOS (e.g. Realtime quotes)
•Push/Pull
•Security (encryption etc.)
determines
•Handle interface changes
•Disable broken connections
•Add new sources
•Poll and re-enable sources
•Keep statistics on sources
Reliability/ Performance
The SAL shields the portal from external data/application sources
Problem analysis

Distribution Architecture

Figure 6.5.



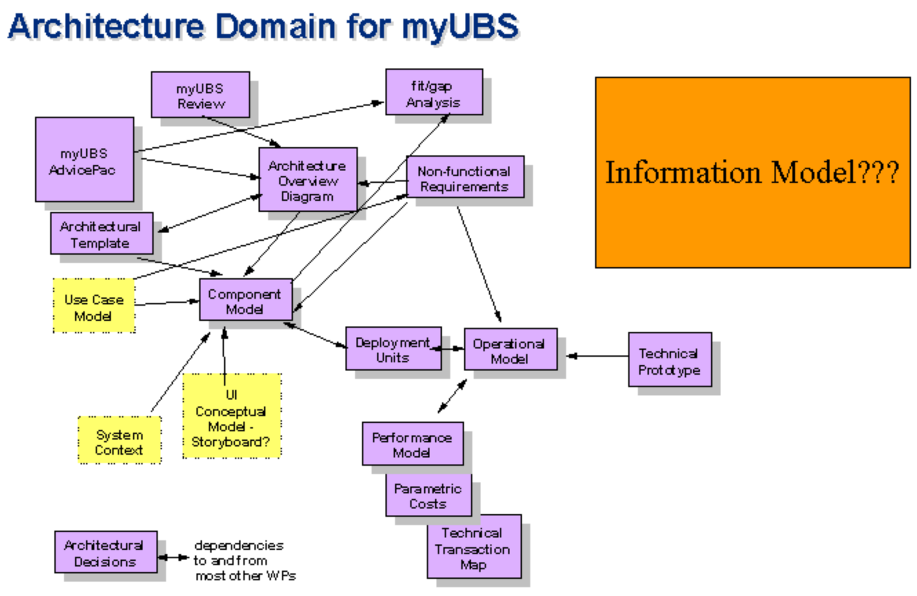
Distribution Architecture

	Source	Protocol	Port	Avg. Resp.	Worst Resp.	Down-times	Load-bal.	Security	Contact/SLA
News	hostX	http/xml	3000	100ms	6 sec.	17.00-17.20	client	plain	Mrs.X/News-SLA
Research	hostY	RMI	80	50ms	500ms.	0.00-1.00	server	SSL	Mr.Y/res-SLA
Quotes	hostZ	Corba/IDL	8080	40ms	25 sec.	Ev.Monday 1 hour	Client	plain	Mr.Z/quotes-SLA
Personal	hostW	JDBC	7000	30ms	70ms	2 times Per week	server	Oracle JDBC dr.	Mrs.W/pers-SLA

Getting this information requires tracking backend services and writing test programs. The results determine what can be combined on a personalized homepage.

Architecture Domains

Figure 6.6.



System Architecture

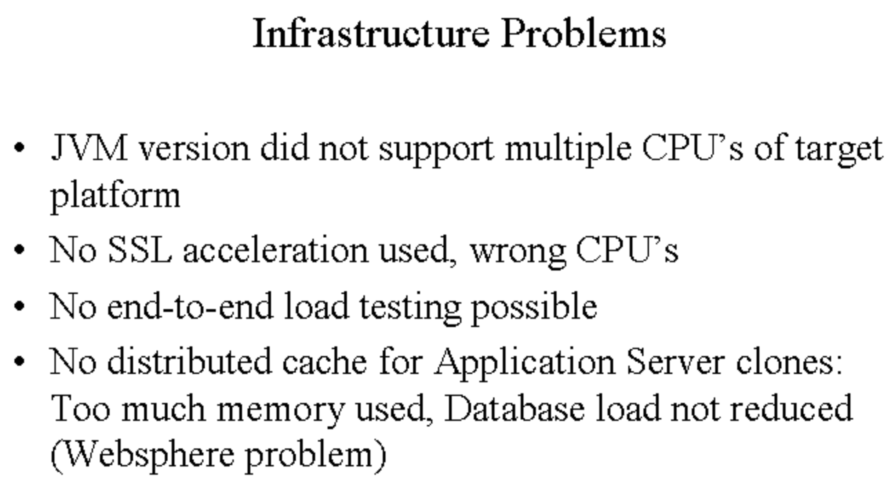
Software Architecture

Physical Architecture

Vertical vs. Horizontal Scalability, high Availability

Infrastructure Problems

Figure 6.7.

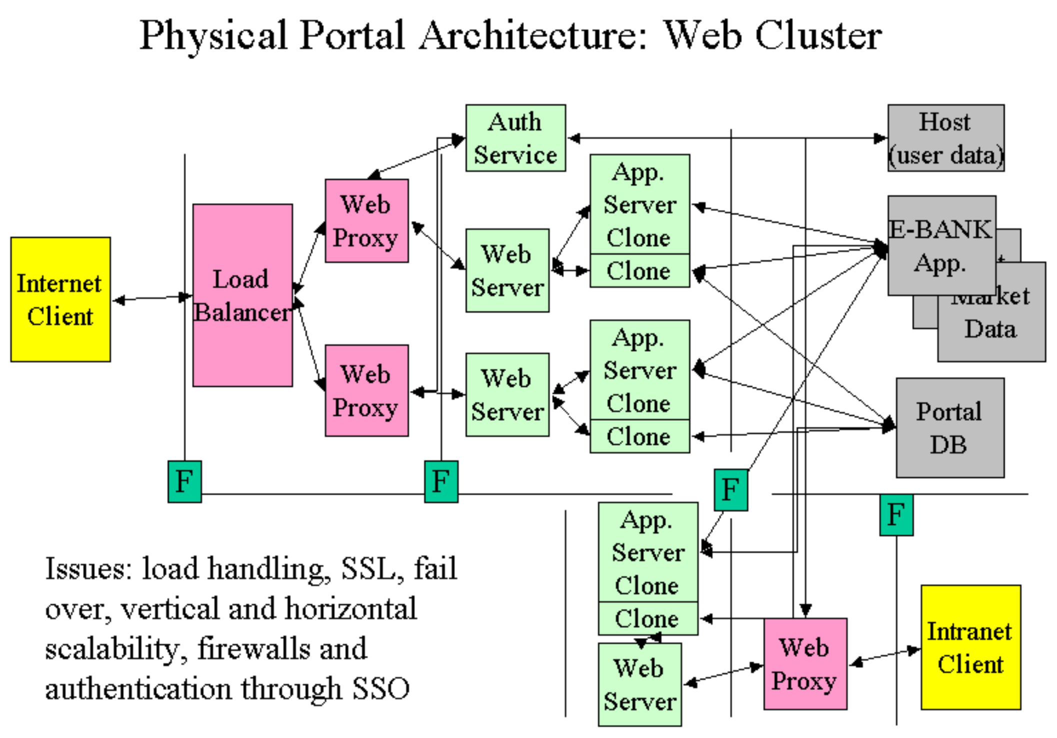


Infrastructure Problems

- JVM version did not support multiple CPU's of target platform
- No SSL acceleration used, wrong CPU's
- No end-to-end load testing possible
- No distributed cache for Application Server clones: Too much memory used, Database load not reduced (Websphere problem)

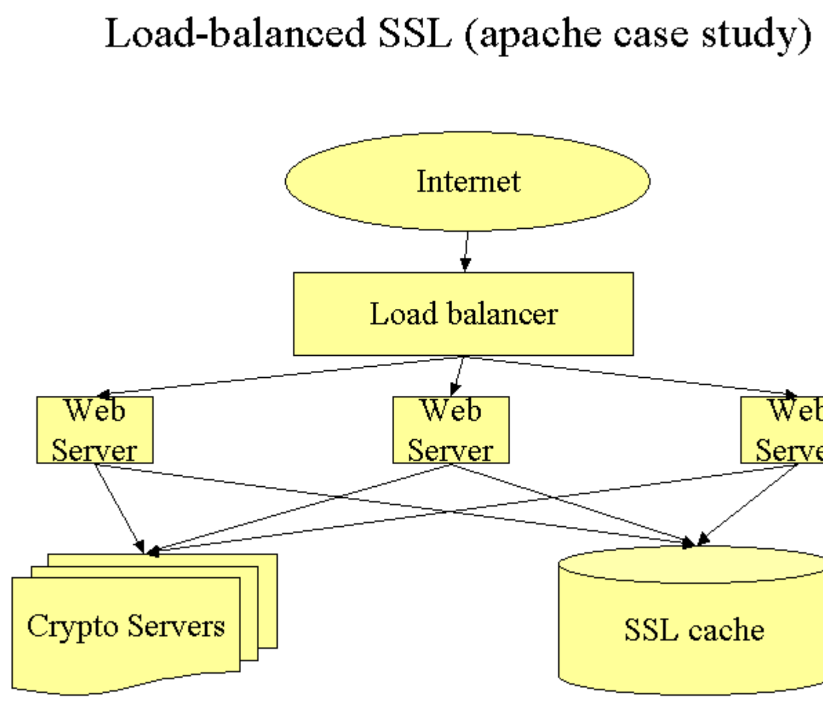
Portal on a Web Cluster

Figure 6.8.



SSL Acceleration

Figure 6.9.



Physical Architecture Options

Distributed Caching

Figure 6.10.

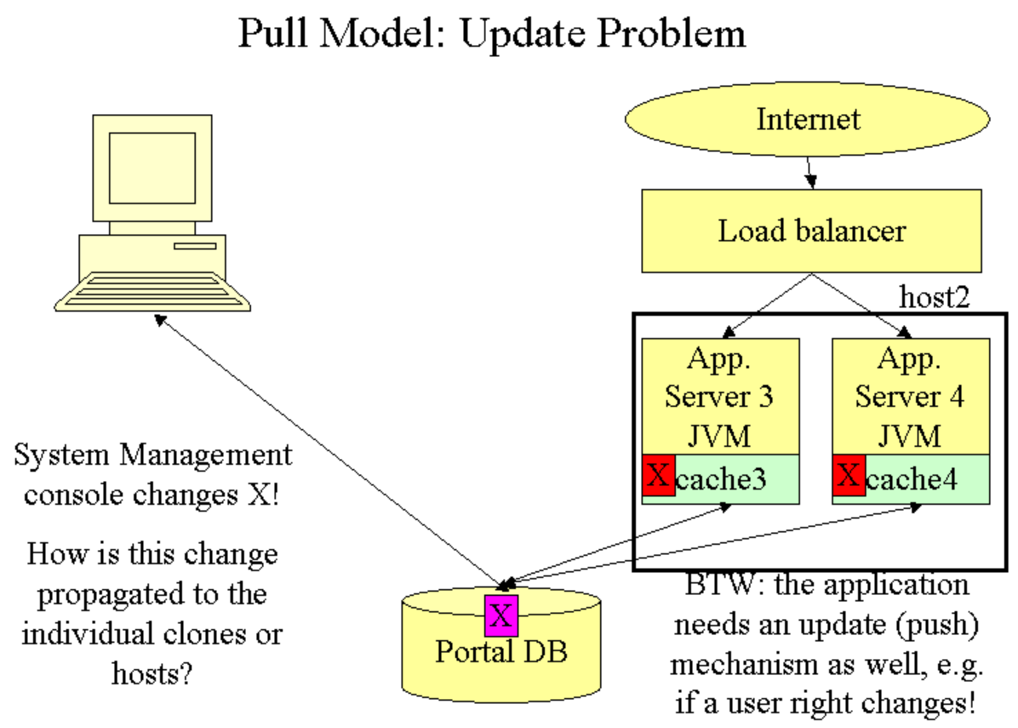

Pull Model: Update Problem
Internet
Load balancer
host2
App. Server 3 JVM
X cache3
App. Server 4 JVM
X cache4
System Management console changes X!
How is this change propagated to the individual clones or hosts?
X
Portal DB
BTW: the application needs an update (push) mechanism as well, e.g. if a user right changes!

Figure 6.11.

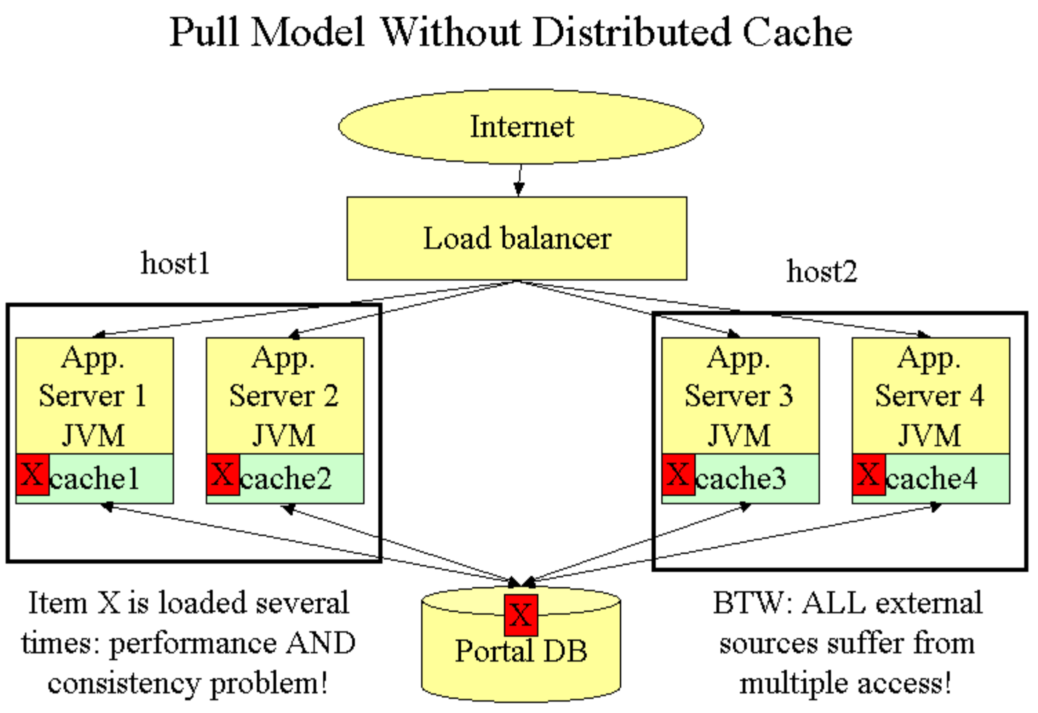

Pull Model Without Distributed Cache
Internet
Load balancer
host1
host2
App. Server 1 JVM
X cache1
App. Server 2 JVM
X cache2
App. Server 3 JVM
X cache3
App. Server 4 JVM
X cache4
Item X is loaded several times: performance AND consistency problem!
X
Portal DB
BTW: ALL external sources suffer from multiple access!

Figure 6.12.

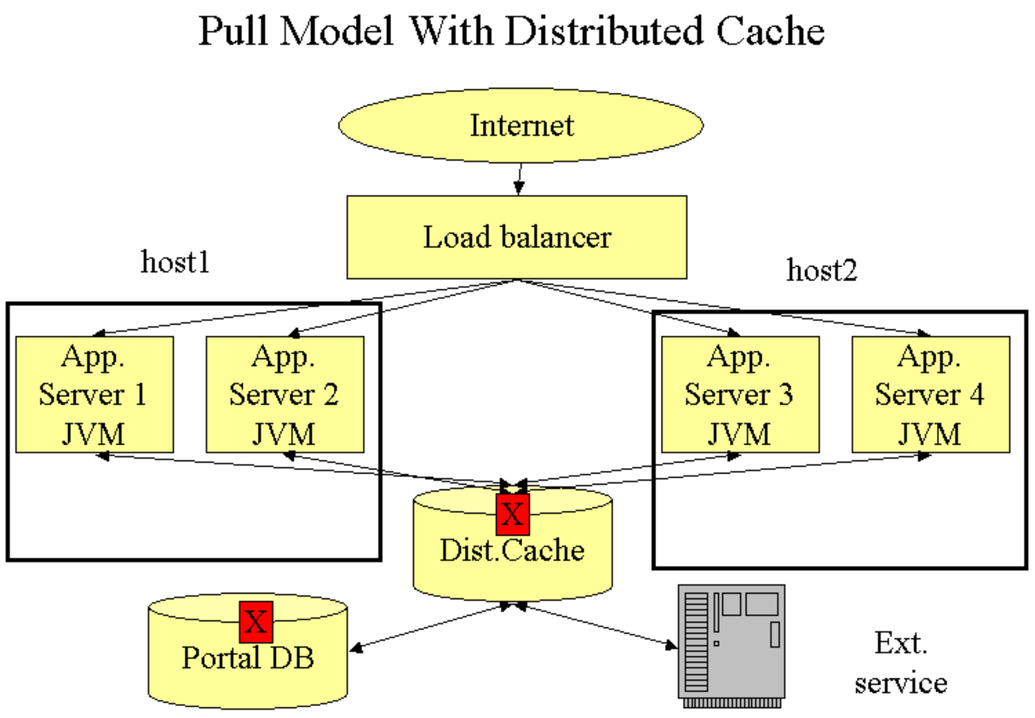
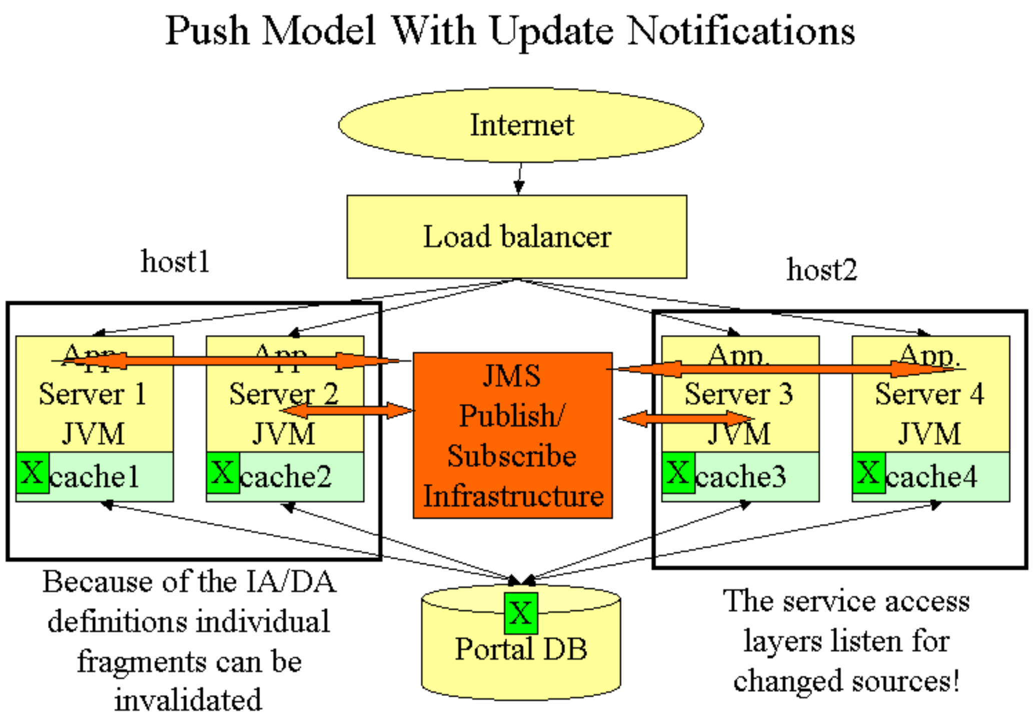


Figure 6.13.


Push Model With Update Notifications
Internet
Load balancer
host1
host2
App. Server 1 JVM
X cache1
App. Server 2 JVM
X cache2
JMS Publish/ Subscribe Infrastructure
App. Server 3 JVM
X cache3
App. Server 4 JVM
X cache4
X
Portal DB
Because of the IA/DA definitions individual fragments can be invalidated
The service access layers listen for changed sources!

Chapter 7. Portal Maintainability

Delegated Management

What is "Delegated management services (DMS)"?

From Netegrity's Siteminder product overview: (36)

"Due to the complexity of distributed portal environments, portal administrators are challenged to administer disparate users and groups of users across multiple organizations, partner and affiliate sites. DMS establishes the distributed hierarchy portal administrators are in search of to ease administration complexities.

Portal administrators can now establish a super administrator who can delegate administration privileges to distributed organizations and organization administrators. Super administrators can enable, disable, modify and move users anywhere within the portal environment and are also able to create organizations and organization administrators for internal departments or external partners and business affiliates. Organization administrators can be granted full administration permission to enable and disable users, and modify user attributes within the organizations they are responsible for or they can be given more constrained access such as to only modify user attributes for example. It's left to the discretion of the portal administrator as to what permissions are granted to the organization administrators. Users in all organizations can be given administration permission to modify all or a select set of their user attributes. The particular set of attributes is left to the administrators managing the organization the user belongs to.

DMS also provides event-driven workflow for pre-process and post-process administration and registration events. Shared workflow libraries can be developed and customized to support the workflow functionality required. Libraries can evaluate a pre-process request and govern whether to accept or reject it. If accepted, the request is carried out. If rejected, the request will not be carried out and a pre-process error will be returned. After a DMS request is successfully processed, workflow libraries evaluate the post-process request and take any action as dictated by the business process and return success or fail status to the DMS application."

Service Access Layer (SAL)

Probably the biggest weakness of AEPortal currently lies in the missing SAL – both at the presentation side as well as when accessing backend services. A typical Service Access Layer composite DesignPattern has recently been published by Oliver Vogel and it could be a starting point for the AE-Portal re-design. (more to come)

Figure 7.1.

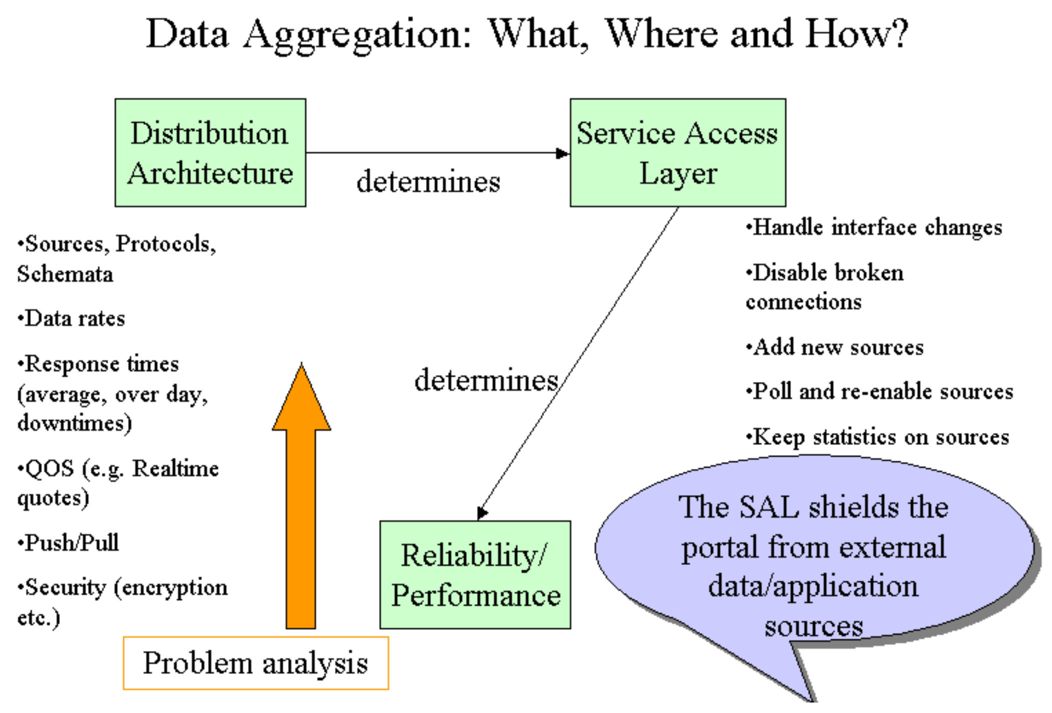
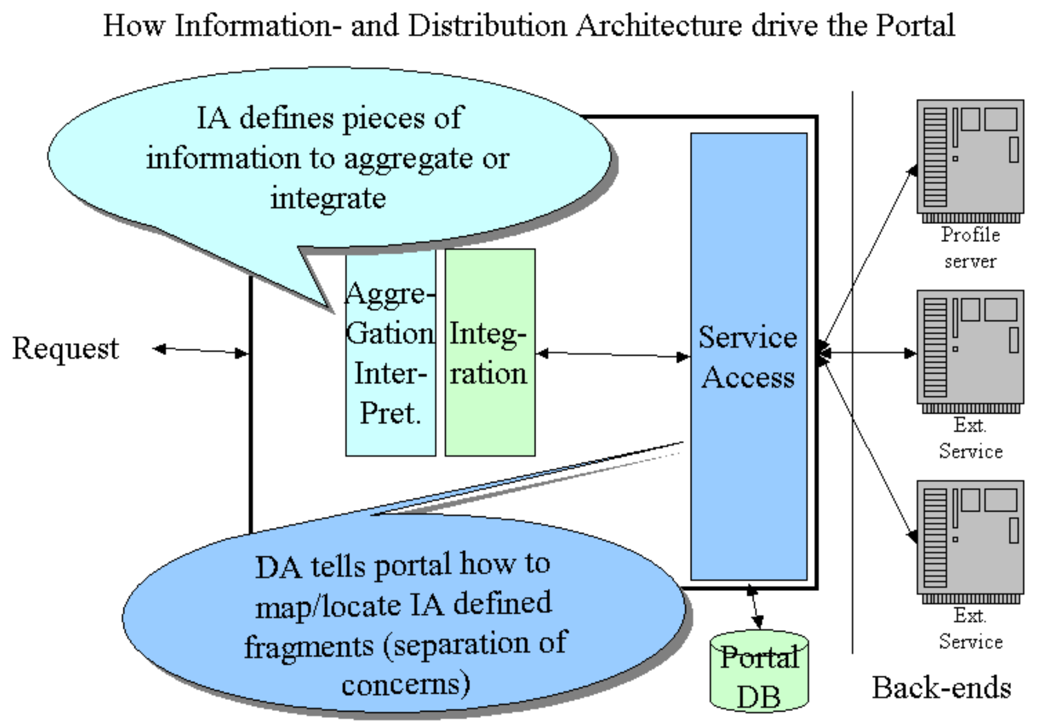

Data Aggregation: What, Where and How?
Distribution Architecture
determines
Service Access Layer
•Sources, Protocols, Schemata
•Data rates
•Response times (average, over day, downtimes)
•QOS (e.g. Realtime quotes)
•Push/Pull
•Security (encryption etc.)
Problem analysis
determines
Reliability/ Performance
•Handle interface changes
•Disable broken connections
•Add new sources
•Poll and re-enable sources
•Keep statistics on sources
The SAL shields the portal from external data/application sources

Figure 7.2.


How Information- and Distribution Architecture drive the Portal
IA defines pieces of information to aggregate or integrate
Request
Aggre-Gation Inter-Pret.
Integ-ration
Service Access
DA tells portal how to map/locate IA defined fragments (separation of concerns)
Portal DB
Profile server
Ext. Service
Ext. Service
Back-ends

Service Development Kit (SDK)

The ability to dynamically add new services without re-deploying the whole portal is an absolute must for an enterprise portal – it is not so important for more specific portals like SEPortal.

What would be necessary to extend the current SEPortal code-base with such a feature?

Let’s discuss the cold spots after a quick look at how it could work!

Dynamically loading a new service

1. A department wants to offer a service on the AEPortal portal. They use the AEPortal SDK to create a service package, consisting of java classes, service description, JSP files.

Part of the SDK are tools that use the Visual Age Java Tool API to create the necessary code frames, helper classes and descriptions. (see Dmitris work)

1. The service package is loaded through the AEPortal service loader – itself a service running in AEPortal
2. The service loader inspects the package, updates the AEPortal configurations with the package description and adds the new java classes to the proper factories. The loader then copies the new JSP files to their destinations.
3. The new service needs to get referenced from existing JSPs. But because these JSP build their list of things to render now dynamically, the configuration changes made the new service references immediately available.

Necessary design changes in SEPortal

The service loader mentioned above is not a design change – it is a new feature. Here I want to discuss real changes.

1. Configuration information can no longer be just files. It should be DB based. The system needs to add to it dynamically
2. Besides configuration information, factories are the most import pieces of a dynamic system. The SEPortal factories in Infrastructure are often static (the definition of the objects they can serve are hardcoded). I would replace the Infrastructure factories (e.g. Entity Finder) with "intelligent factories" from Apaches Turbine.
3. The fragment approach discussed above would force us to a more descriptive page building process anyway and we would be able to assemble a page from fragments dynamically.
4. The implementation of authorization via static access tokens per role needs to change in order to support dynamic extensions to existing user types and access tokens. A new service will generally require new access tokens for certain roles. The current design is flawed because it holds all access tokens of a user in a table instead of constructing the access tokens dynamically e.g. a stored procedure into a view.

Assessment

The efforts to build a SDK are considerable but not extreme. I guess we could have something running in 2 month (2 developers)

The question really is IFF we want to build it or use the packaging mechanisms of J2EE. Would they be sufficient? Granular enough? Does it work already?

A prototype of the new and extensible access control framework already exists.

Enterprise Portal: Can there only be ONE?

Before we can decide on this we need to clarify the concept of "portal" a bit better. The one portal approach can mean:

- a common source code base and a single instance of the portal runtime
- a common source code base and different instances of the portal runtime
- different source code bases and different instances of the portal runtime

At the beginning of the AEPortal portal project everybody was quite convinced that a single source code base and runtime instance was the right approach. Services should not know about customer segmentations (because this can change quickly).

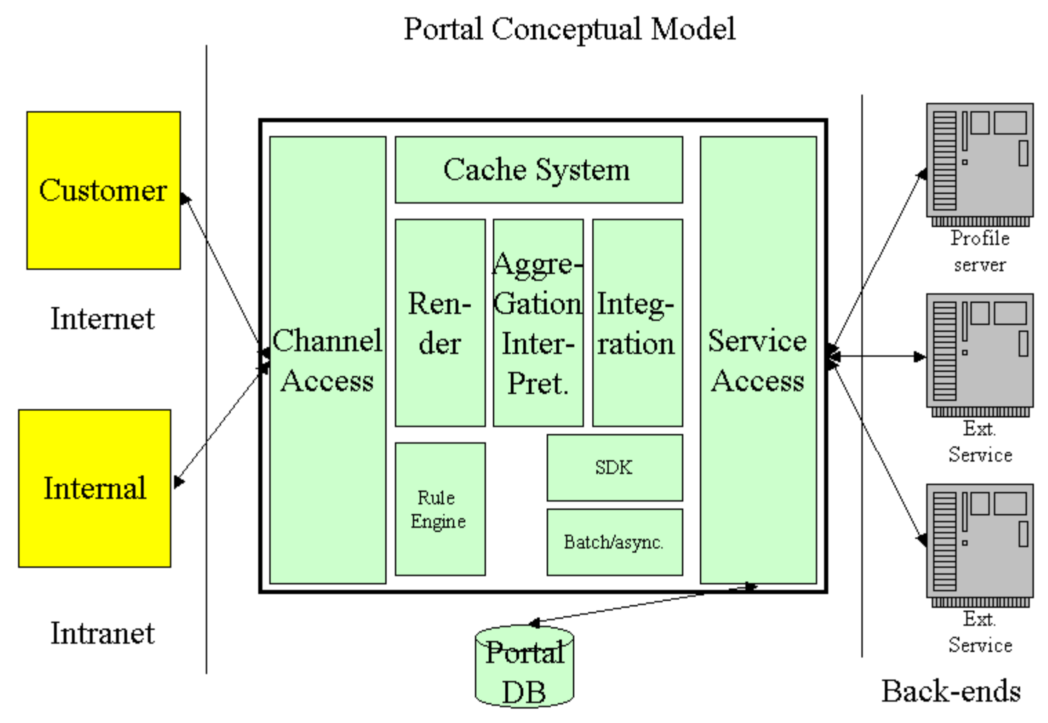
THE LAST FIVE MONTH HAVE SHOWN THAT WE UNDERESTIMATED THE DIFFERENCES IN ARCHITECTURE AND IMPLEMENTATION CAUSED BY THE DIFFERENCES IN NON-FUNCTIONAL REQUIREMENTS.

•Is it clever to have only one instance of a Portal for a large Enterprise? (update problem, QOS for special customers)

•What is the price of having only one code-base? (missed time to market, missed optimization, missed functionality, missed opportunities)

While the conceptual model of a portal is quite simple, the non-functional requirements can lead to many differences in architecture and implementation.

Figure 7.3.



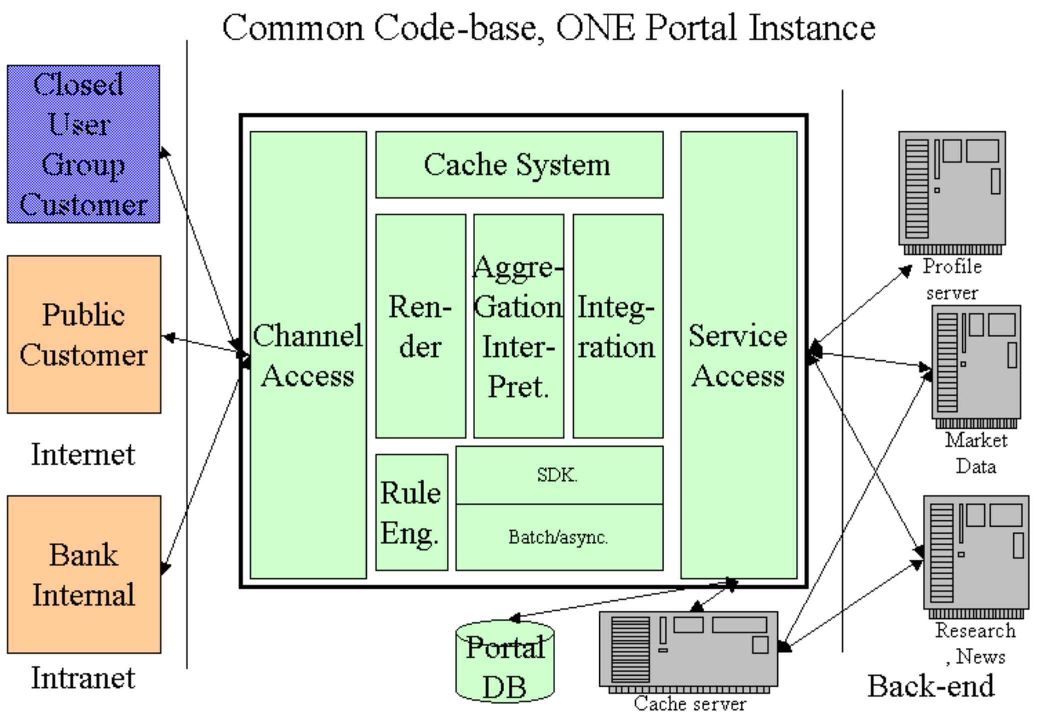
The Portal conceptual model:

• Must contain the basic building blocks

• Real implementations need to specialize the conceptual model with respect to scalability (batch, caching, SDK) or special requirements (rule engine)

One source-code base and only one installation of THE Enterprise portal.

Figure 7.4.



Benefits	*Requirements*
Standard coding practices	Requires a module concept that allows all departments to integrate their parts
Common framework, re-use and efficiency	Requires a service development kit supporting this
Robust and scalable services	Requires implementations to support large scale and high-speed operation
Integration of enterprise information systems	Requires backend services to scale enormously Changes to backends (e.g. External Data System) needed
Centralized operating	Integration into operations infrastructure
Common services used (authentication, authorization)	Integration into service infrastructure (authentication and authorization services etc.) Wait for those services to complete and scale

Problems:

• Hard to guarantee Quality of Service for special customers

• Upgrades are hard and dangerous!!

• Upgrades to individual components are tied to general release plan!

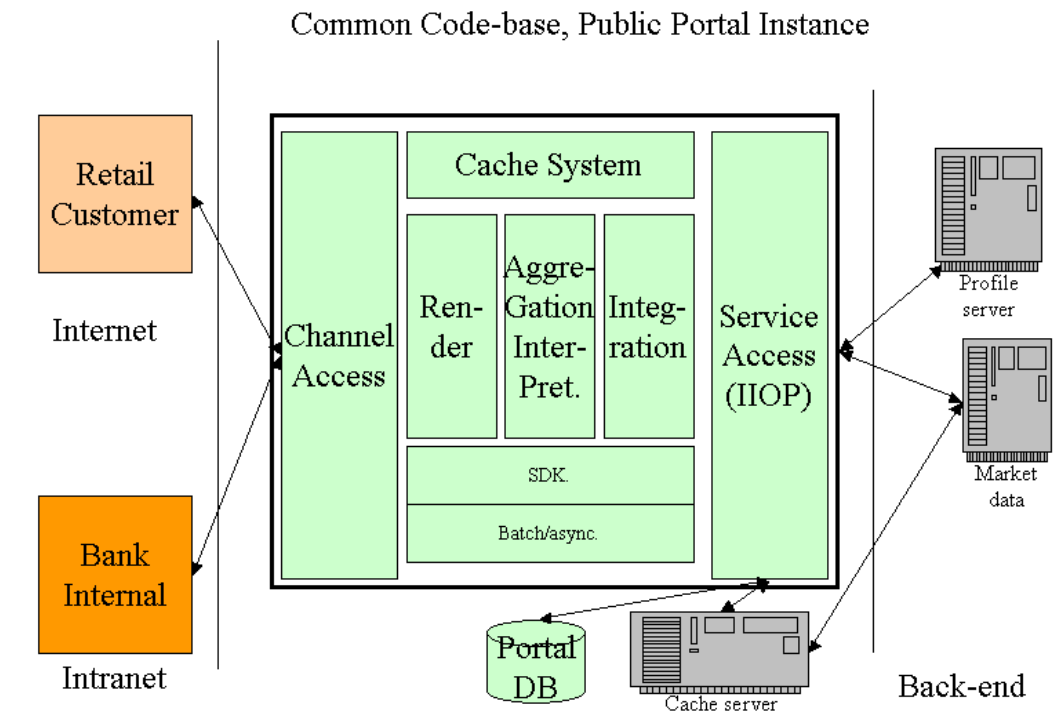
My guess: It won't work!

one source-code base, different installations and configurations.

Not much different from a) besides duplicated operating needs. The biggest benefit is that it is easier to guarantee a certain quality of service for special customers and to make installations more flexible

Benefits	*Requirements*
Different installations can be upgraded independently	Need to deal with different releases. Duplicated operating.

Figure 7.5.



Problems of the Retail Portal:

• Implementation must work on a much larger scale

• Implementation requires different architecture

• No rule engine (performance). Static template based rendering

• High-speed profile server necessary

• High-speed cache server necessary

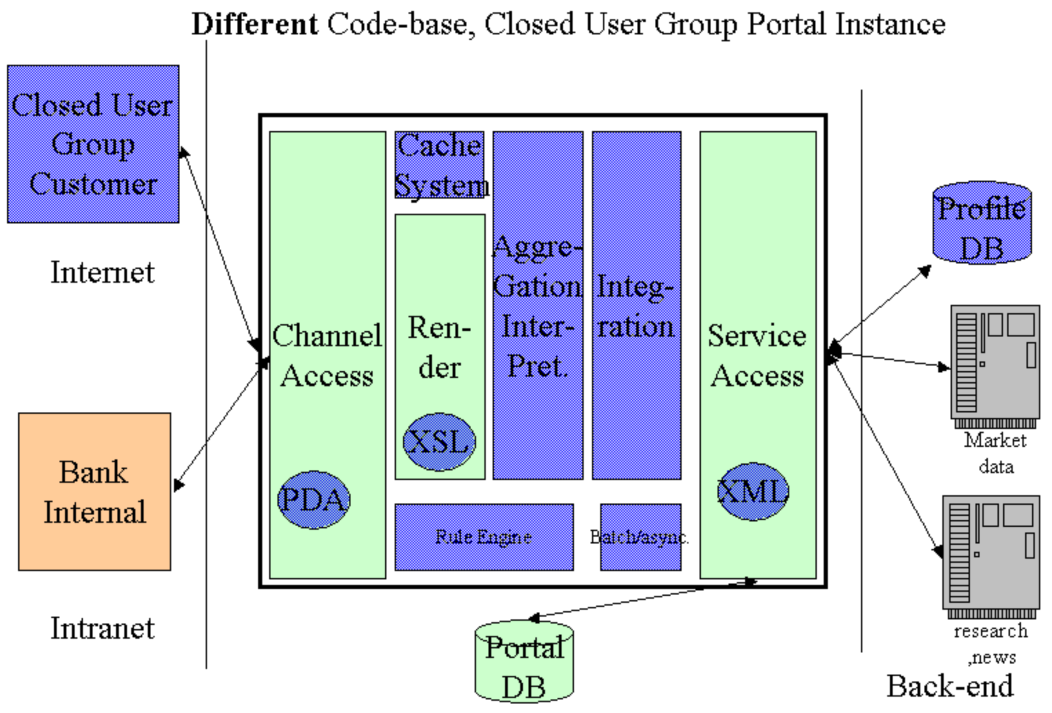
The common code base would have to follow the retail portal requirements first - because of the scalability problems.

different source-code bases and different installations

This list is easy to create: take the table from a) and put a NOT in the requirements column.

Benefits	*Requirements*
Authorization, page integration etc. will NOT be able to integrate services from other departments automatically	NOT: Requires a module concept that allows all departments to integrate their parts
No common framework, re-use and efficiency.	NOT: Requires a service development kit supporting this
Services scale to real need	NOT: Requires implementations to support large scale and high-speed operation
No Integration of enterprise information systems	NOT: Requires backend services to scale enormously. NOT: Changes to backends
NO Centralized operating, Quick deployment!	NOT: Integration into operations infrastructure,
Common services used (authentication, authorization)	NOT: Integration into service infrastructure (ESAUTH,BBS etc.) NOT: wait for those to complete But the integration can be an OPTION!

Figure 7.6.



Clearly alternative a) sounds the most attractive. It serves all the buzzwords of re-use, framework, scalability etc. The only problem is that AEPortal/SEPortal might show that it simply does not work and that it put EVERYBODY involved into a disadvantage!

Note: SEPortal is much later because of the integration into the large-scale site AEPortal. The large-scale site AEPortal itself is later because some of the technology used for SEPortal does not scale for AEPortal. The reason for this is simply the non-functional requirements which are very different for both portals!

The SEPortal Portal can:

• Live with a simpler architecture because of fewer scalability problems

• Does not need SDK. Needs less caching and batch processing.

• Rule engine possible (fewer user). Advanced XSL based rendering, better integration and aggregation. In general, a small to medium scale portal can serve as a test-bed for new technology that wouldn’t work right away on a large scale.

• No high-speed profile server necessary

• No high-speed cache server necessary

• Faster time to market.

• No need to change back-ends (MADIS etc.)

The Usability tests conducted recently may even force us to recognize fundamentally different user behavior (e.g. much longer sessions for External Asset Manager and other specialists, a much higher degree of customization for those expert groups).

Note: the Servlet API 2.1 introduced HttpSession.setMaxInactiveInterval(int interval). This MIGHT allow a specific timeout PER session.

See: www.javaworld.com/javaworld/jw-12-1998/jw-12-servletapi_p.html [http://www.javaworld.com/javaworld/jw-12-1998/jw-12-servletapi_p.html] (Jason Hunter)

Recommendation: SEPortal the template for smaller portals

With some improvements SEPortal could be the implementation base for many small to medium portals in a large enterprise while the retail portal is developed using a new architecture supporting large-scale use and extended backend services.

Chapter 8. Content Management Integration

The portal architecture as described above does not include a Content Management System (CMS) yet. The reasons are two-fold:

- Integration problems (e.g. security, programming language)
- different skill sets (Java Programmers vs. Document People using scripts etc.

The first try to integrate a CMS ran into problems in the areas of security and programming language. The goal was to use the CMS for "content type" things in the portal, e.g. FAQs and messages. Unfortunately the API of the CMS used Perl and the portal itself used Java. This was not the only problem: The portal used instance based authorization while the CMS (especially the authoring subsystem) used a type based authorization.

Example 8.1. Authorization between portal and CMS

Most CMS require every piece of content to have a "type", e.g. "message" with an associated schema describing it. The authorization subsystem of the CMS can only associate either types with roles or specific instances of types with roles. In the case of type based authorization every member of a certain role gets access to all instances of a content type - not what a portal usually requires e.g. if messages should be private between a client it ITS advisor. On top of this, authentication mechanisms need to be compatible between portal and CMS as well.

Why use a CMS?

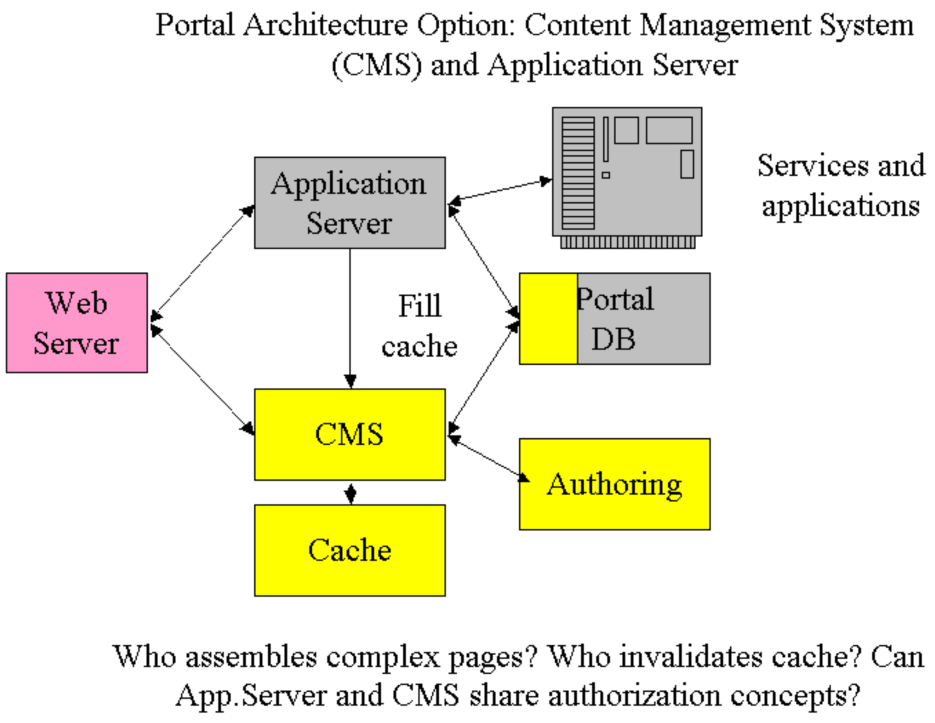
The experiences with a portal built on Application Server technology alone have shown that a CMS has certain advantages:

- better handling of content fragments and caching. Caching needs to be built from scratch in an application server based portal.
- a complete authoring environment for pure content (e.g. FAQ, messages)
- a permission based automatic workflow and release process allowing business users to control the portal content to a certain degree. This is not possible in a JSP based personalized portal.
- a defined way to add meta-information to pages (later used to better mine site stats.
- a search facility
- usually also automatic separation of navigation and content, supported by tooling.

Architecture

Several possible ways to use a CMS for or within a portal come to mind. The first and most important question is: who gets a request first - CMS or application server? And who assembles the results into the firnal form? And how is the content coming from the application server represented in the CMS (tooling etc.)? Does the application server deliver only presentation-less results, e.g XML streams? Or do both, CMS and application server extract results in XML and then somebody assembles the results to a personalized page? This would circumvent the advanced caching capabilities of the CMS.

Figure 8.1.


Portal Architecture Option: Content Management System (CMS) and Application Server
Application Server
Services and applications
Web Server
Fill cache
Portal DB
CMS
Authoring
Cache
Who assembles complex pages? Who invalidates cache? Can App.Server and CMS share authorization concepts?

Chapter 9. Heterogeneous and Distributed Searching

Because auf its dynamic nature a personalized portal that integrates applications and various back-end services in realtime has its problems providing a top level search facility. Much of the discussions in this chapter draws from an article on the future of internet search by Axel Uhl.

The surface web and the deep web

Uhl differentiates between static web pages (surface web) and dynamically generated pages (sometimes within a session context) or dynamic queries (deep web). Regular search engines cannot access content in the deep web, because the content returned from HTTP POST requests is not indexable (it does not have a URL). This content grows at a frightening rate and is already now more than 500 times bigger than what's available on the surface web. Uhl suggests applications to offer a query interface that can be used by a search framework to map a top level query to different underlying applications and data sources. The keywords here are heterogeneous and distributed search.

From an enterprise portal point of view it would be nice being able to offer a) a top level global search across all services b) a site directory, generated, that allows browsing type access to all information

There is of course the problem of mapping a fragment based architecture to a search mechanism. Here so called "topics" - kind of "canned queries" could offer a solution. But the biggest and still unsolved problem is the definition of the information model for the portal.

A different problem is performance: In the chapters above we have shown how backend access affects performance negatively. Offering a search mechanism can easily conflict with the approach of minimizing backend access. So where does a top level search really work from? The performance problem mirrors the one Uhl has diagnosed for internet search in general: a centralized index causes bandwidth problems (content has to come to the search engine) and performance problems (the query itself is not distributed to the sources and the sources cannot work on it concurrently). Last but not least do we need a mechanism to cache query results (see the IBM Watson paper on this topic).

Architecture

Uhl suggests an object oriented architecture with the following interfaces:

Figure 9.1.

This would result a portal search architecture:

Figure 9.2.

Chapter 10. Mining the Web-House

This chapter totally reverses our point of view: from building a portal to sell services and products to collecting information about customers during their visits. Both views are - and that is one of the greater problems - highly connected: A portal is at the same time an offer (information, services or products) and a measuring tool. But the measurements are again dependent on the content that is offered by the portal - supposedly adjusted after the measurments have been analyzed and the results have flown back into the portal. But the two roles of a portal are very different with respect to the technology used and the persons involved: The "output" side is all real-time and driven by business ideas. The input side needs much more time for analysis and is usually done by special analysts and specialists for data-mining and data-warehouses.

Purpose

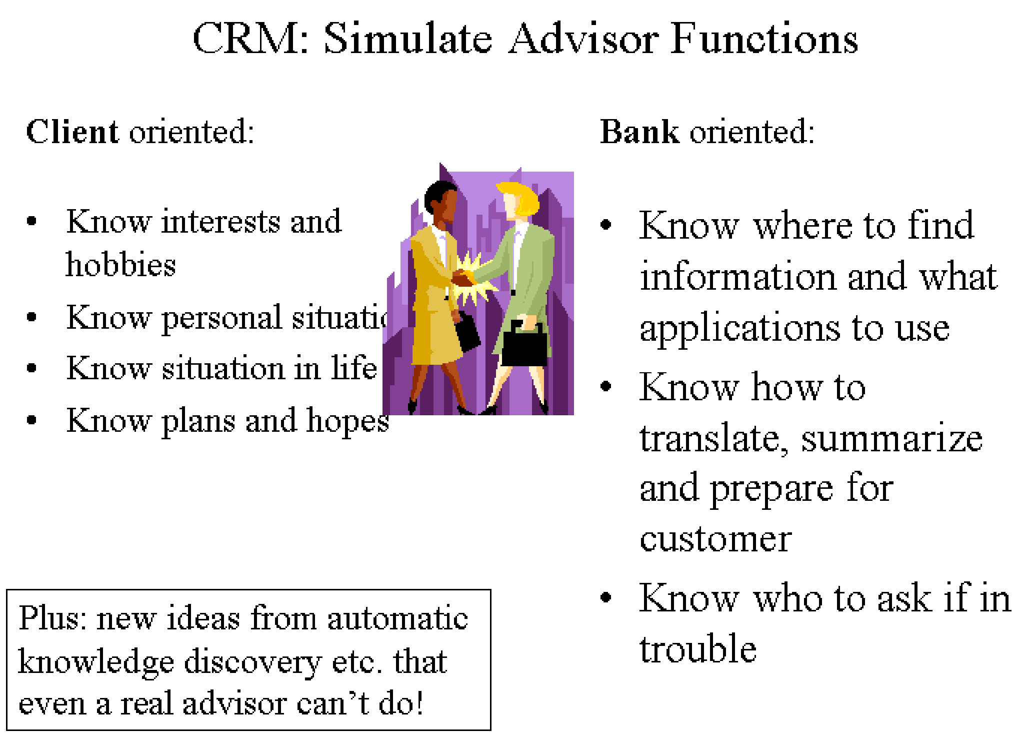
The purpose of this chapter is to clarify the why and how of integrating a data-warehouse with the enterprise portal. Not to provide an introduction to data-warehouses or data-mining in general. Examples of analytics run in the warehouse serve only the purpose to define informations needed from the portal.

But before diving into the technicalities it pays to think about the reasons for the increasing interest in "web-shouses".

Example 10.1. An artificial advisor

The banking business has always been service intensive. The banking personnel knew their customers (this was actually the base of their credit business) and at the same time the personnel also knew how the bank worked internally. They knew the applications and how to use them to extract information for customers.

Figure 10.1.



CRM: Simulate Advisor Functions

Client oriented:

- Know interests and hobbies
- Know personal situati
- Know situation in life
- Know plans and hopes

Plus: new ideas from automatic knowledge discovery etc. that even a real advisor can't do!

Bank oriented:

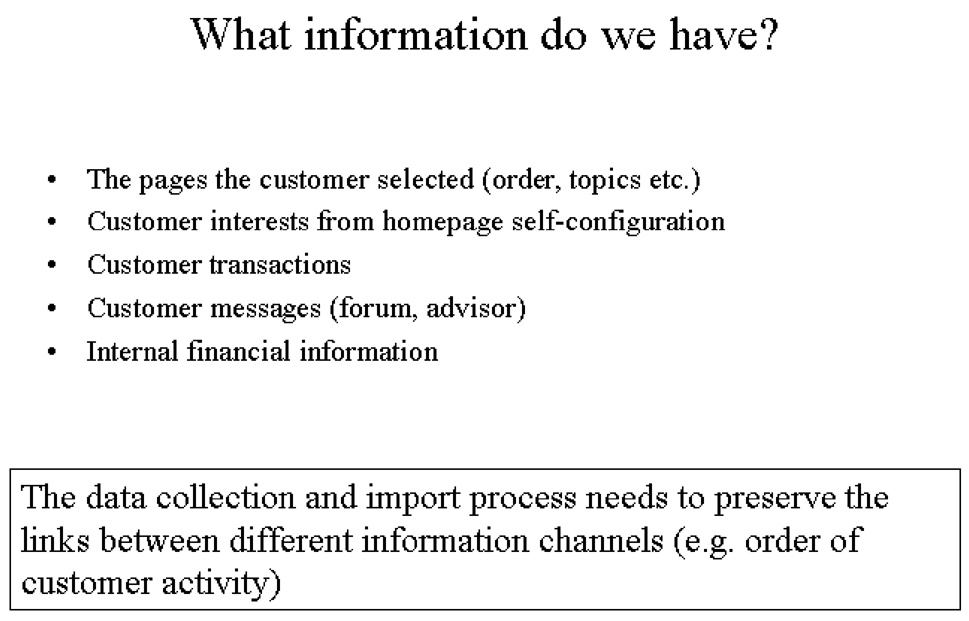
- Know where to find information and what applications to use
- Know how to translate, summarize and prepare for customer
- Know who to ask if in trouble

Nowadays banks need to cut down on personnel costs and one way to do so is to provide their services through a customer friendly (read personalized) portal. We can learn about the requirements of such a portal from the functions that were traditionally provided by client advisors. These functions can be split into two different areas: Learning information about the customer for input into the banking systems. Using banking systems to extract, transform and aggregate information for the customer. Both areas together form what business calls *Customer Relationsship Management (CRM)*

The Portal as an information source

The information that can be collected through a portal is a direct result of what the portal provides in its User Interface. Behavioral information (e.g. clickstream data about page impressions) but also transaction information (e.g. orders) or information coming from collaborative services (e.g. forum activity). Customization performed by a user typically are indicators of special interests (e.g. setting filters for news or research). A search interface also provides information about interests.

Figure 10.2.



What information do we have?

- The pages the customer selected (order, topics etc.)
- Customer interests from homepage self-configuration
- Customer transactions
- Customer messages (forum, advisor)
- Internal financial information

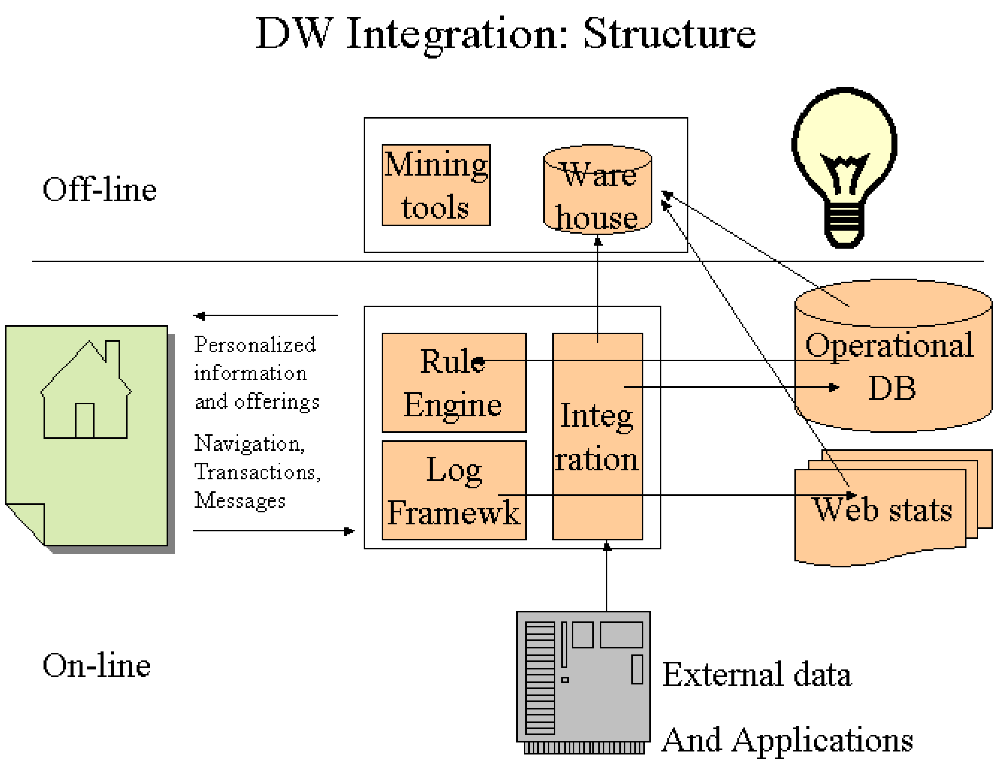
The data collection and import process needs to preserve the links between different information channels (e.g. order of customer activity)

An important question in an enterprise portal is typically whether the users have to be authenticated or not. In case of no authentication it is much harder to personalize a site during the progress of a session because of the realtime requirements and the lack of information. Market-basket analysis in supermarkets sometimes tries to deduce single customers from shopping sequences using e.g. "phenomenal information".

Architecture

A rough sketch of a portal-warehouse connection:

Figure 10.3.



Collecting data: logging and the role of meta-data

Analytics

Figure 10.4.

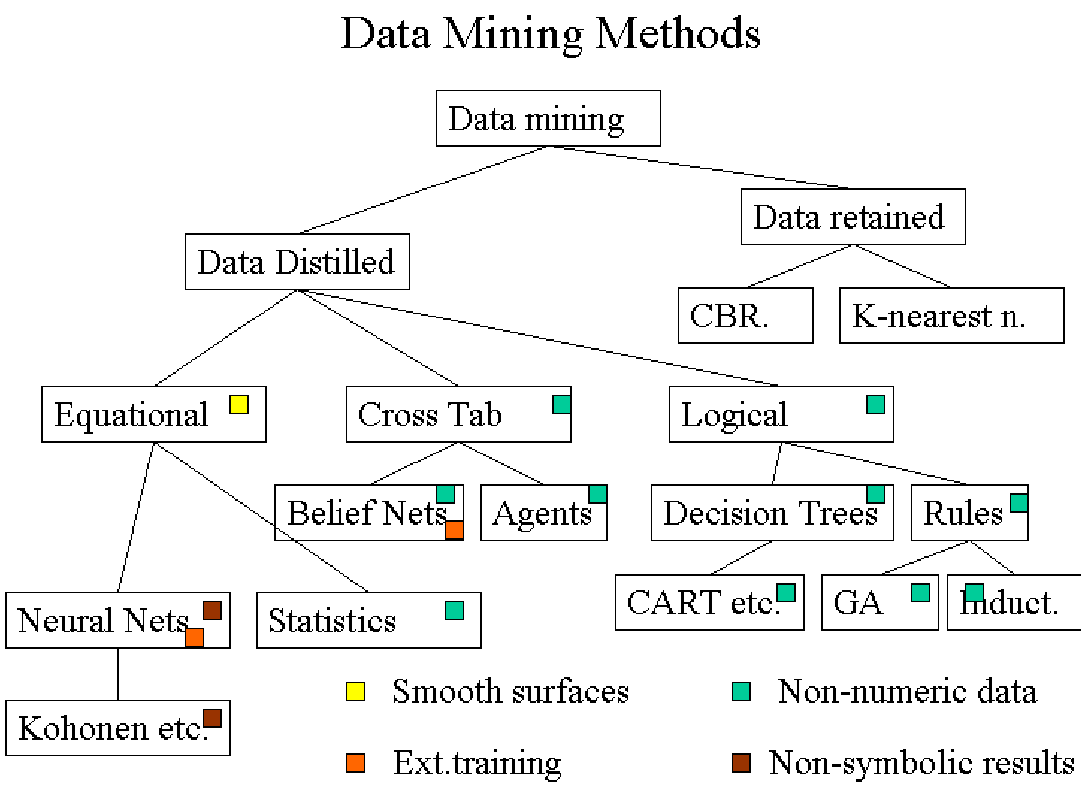
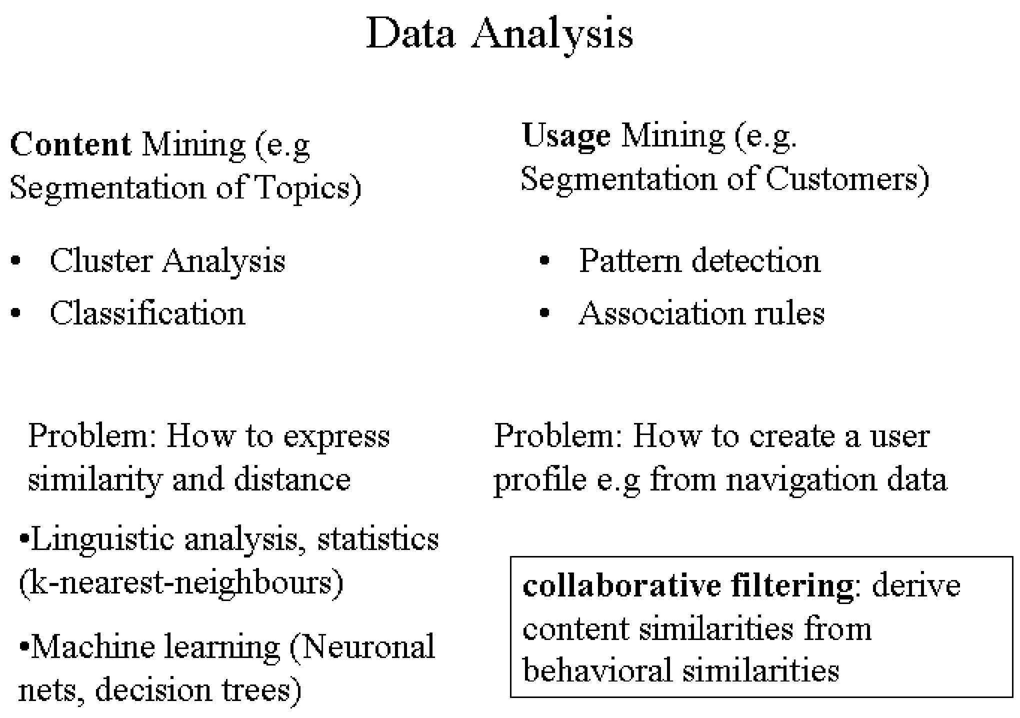

Data Mining Methods
Data mining
Data Distilled
Data retained
CBR.
K-nearest n.
Equational
Cross Tab
Logical
Belief Nets
Agents
Decision Trees
Rules
Neural Nets
Statistics
CART etc.
GA
Induct.
Kohonen etc.
Smooth surfaces
Non-numeric data
Ext.training
Non-symbolic results

Figure 10.5.



Data Analysis

Content Mining (e.g Segmentation of Topics)

- Cluster Analysis
- Classification

Problem: How to express similarity and distance

•Linguistic analysis, statistics (k-nearest-neighbours)

•Machine learning (Neuronal nets, decision trees)

Usage Mining (e.g. Segmentation of Customers)

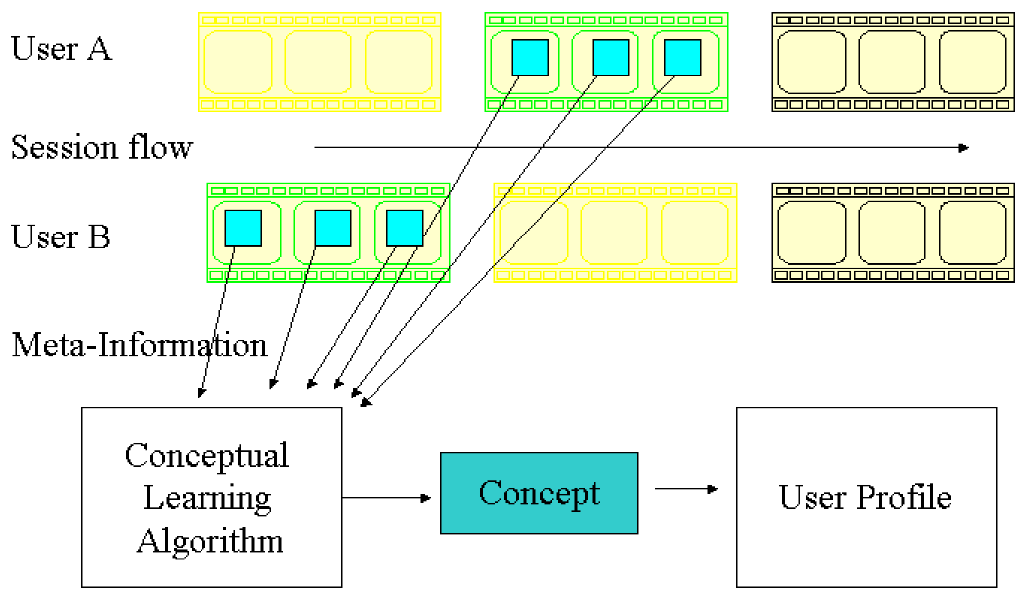
- Pattern detection
- Association rules

Problem: How to create a user profile e.g from navigation data

collaborative filtering: derive content similarities from behavioral similarities

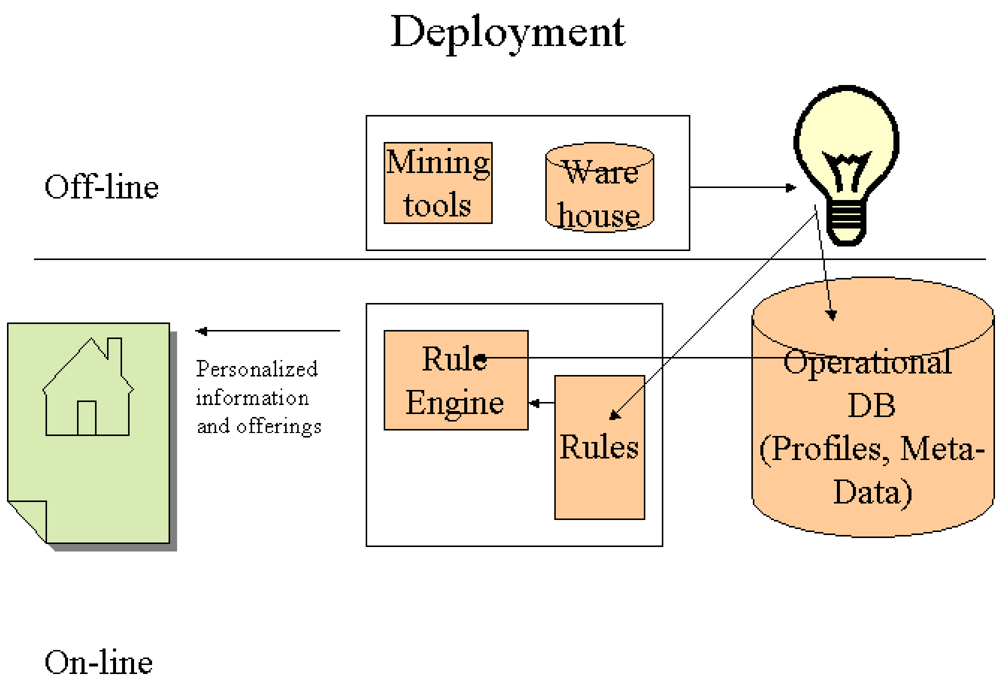
Figure 10.6.

Learning Concepts



How to flow the results back

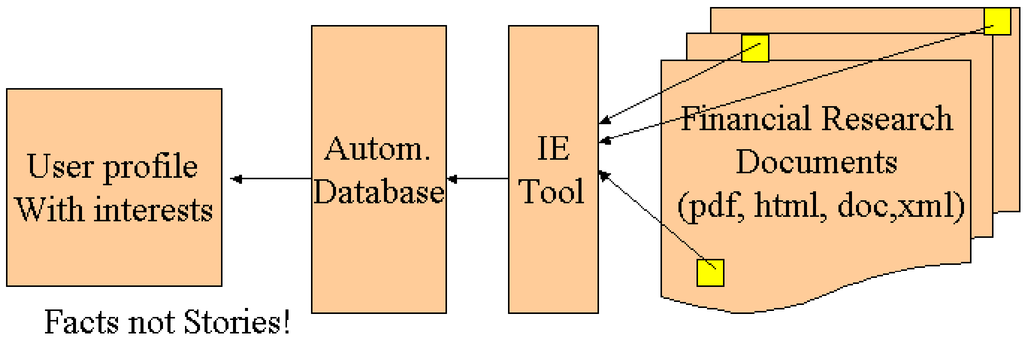
Figure 10.7.


Deployment
Off-line
Mining tools
Ware house
Personalized information and offerings
Rule Engine
Rules
Operational DB (Profiles, Meta-Data)
On-line

External Information Sources

Figure 10.8.

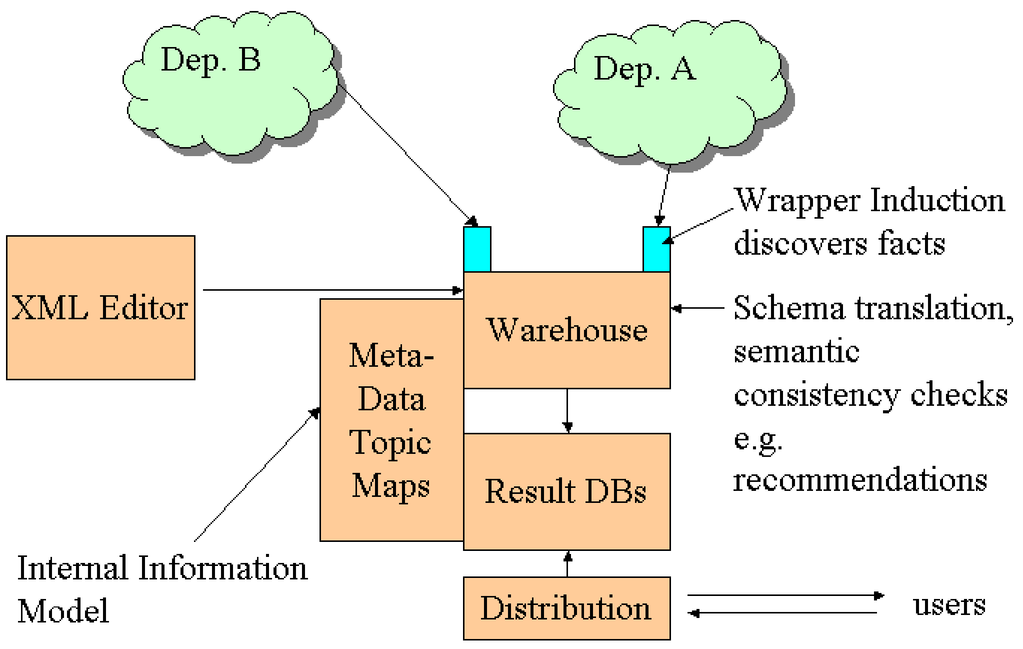
The Text-Warehouse: Information Extraction



Serving personalized information requires fine-grained extraction of interesting facts from text bodies in various formats

Figure 10.9.

Financial Research Integration


Dep. B
Dep. A
Wrapper Induction discovers facts
XML Editor
Warehouse
Schema translation, semantic consistency checks e.g. recommendations
Meta-Data Topic Maps
Result DBs
Internal Information Model
Distribution
users

Chapter 11. Portal Extensions

Integrated Information

big upcoming feature is Topics. These are basically canned queries, similar to "custom views" in some email packages or saved searches from a search tool. In addition to storing Favorites, which are just static links within a site, a member could have a customized topic that might be "Show all NewsItems about the CMF that were posted in the last two weeks". This gets turned into an optimized Catalog query that is always live. This is a powerful personalization feature, and proves even more useful in some other Portal types described below.

Integrating heterogeneous databases

Combining Topic Maps and Holonomic Object technology Nan Kevin Gelhard, Summit Racing Equipment, & Paul Prueitt, OntologyStream Heterogeneous databases can be integrated using topic maps as the controlling interface and a Holonomic Object as generalized database. These two technologies are used together to connect a current commercial data and procedural environment to a prototype B2B/B2C portal. The topic map is used as an "observational data warehouse" for legacy databases, while Holonomic Object Technology provides fast, reliable transformations between legacy databases and topic maps. Within highly ramified buyer/seller events, the portal answers such questions as "What fits my need? Is it in stock? If not, when is it expected? What is the selling price?"

Topics

Figure 11.1.

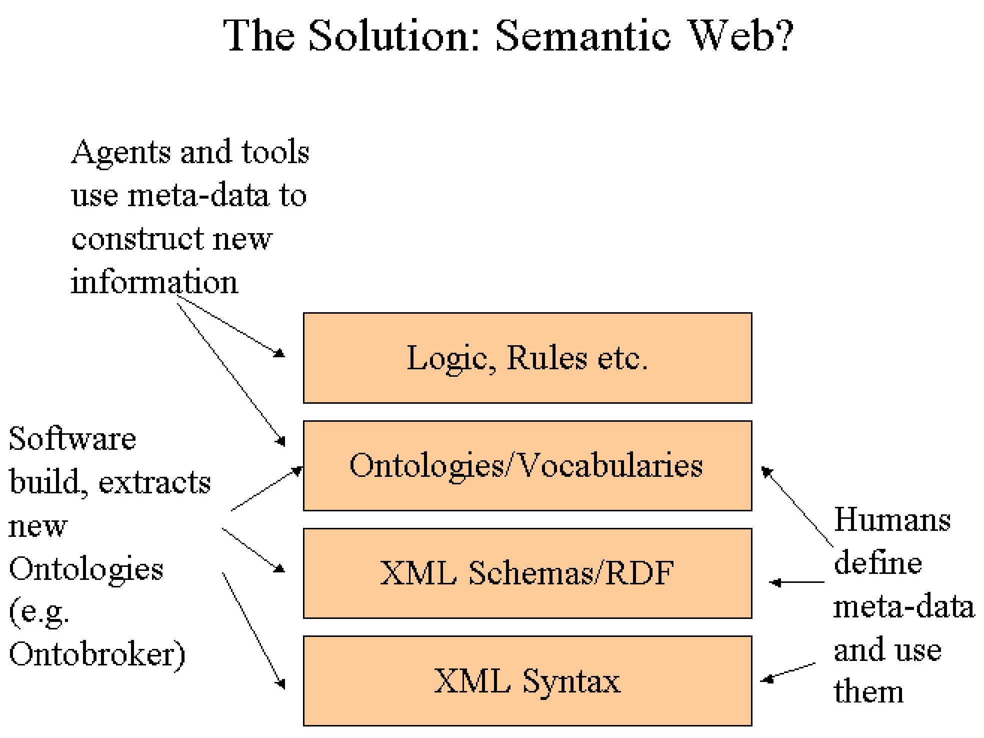
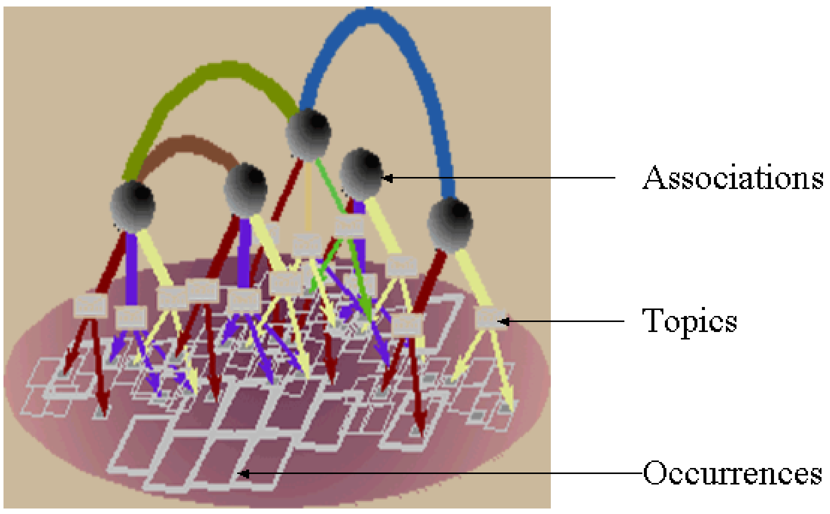

The Solution: Semantic Web?
Agents and tools use meta-data to construct new information
Logic, Rules etc.
Ontologies/Vocabularies
XML Schemas/RDF
XML Syntax
Software build, extracts new Ontologies (e.g. Ontobroker)
Humans define meta-data and use them

Figure 11.2.

AI on Topic Maps?



See: James D.Mason, Ferrets and Topic Maps, Knowledge Engineering for an Analytical Engine

Internet Conversation

Forum

Instant Messaging

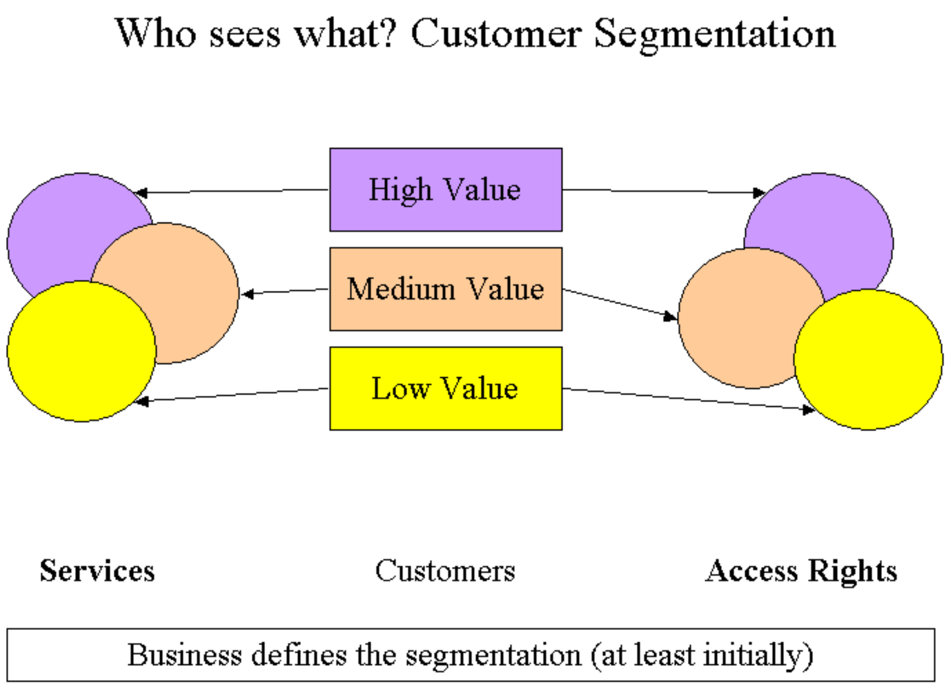
Federated Portals?

Chapter 12. Personalization

Customer Segmentation

One of the most important concepts of an Enterprise Portal is the definition of customers and their segmentation. "Who sees what?" is defined by the segmentation which is usually defined by the business - at least initially. The segmentation defines not only what can be seen or used by a specific customer. It drives access control as well. In other words: Authorization information drives GUI and access control and is itself driven by the customer segmentation.

Figure 12.1.


Who sees what? Customer Segmentation
High Value
Medium Value
Low Value
Services
Customers
Access Rights
Business defines the segmentation (at least initially)

The problems with customer segmentation are as follows:

- Business likes to change the customer segmentation frequently as a result of changes in marketing strategy.
- TheGUI is affected by different segmentations
- Access control is affected by different segmentations
- Customer "types" easily end up being hard-coded
- A rule engine would let business change their definitions without changes in code
- How does one integrate a rule engine with a clean interface and without distributing calls to the rule engine throughout the system?

- Dynamic segmentation needs to perform well. Hard to achieve with rule engines.

Design for change and exception

After the first discussions with business about the portal you will notice that there are few areas that are more discussed than the way customers are categorized. In other words: customer types. The only sensible conclusion coming from those discussion is that the portal needs to define customer segmentation as a "hot spot", designed for change. A more subtle problem are exceptions to rules. Many people like hierarchical, non-overlapping classifications. E.g. a tree of customer types each with non-overlapping access rights. But this is "software" thinking - not business thinking. It should always be possible to also cover the exception like "this is a customer of type A, thereforeshe has rights 5,7,9 but in case it is Mrs.Doe she also gets right 12". In other words: make your customer segmentation not strictly type based. Include instances as well. Do not force the system to create a new type for the above case.

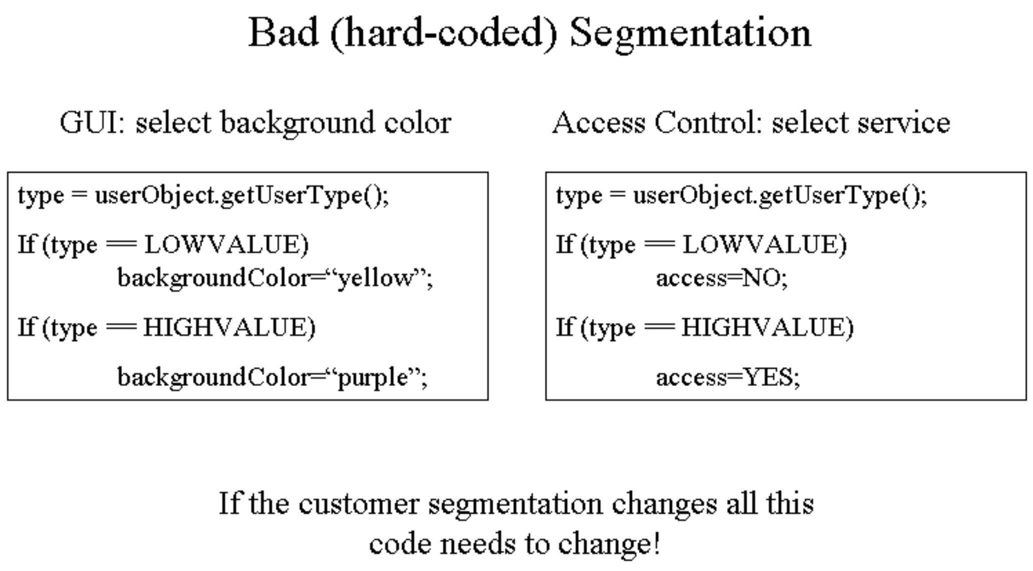
Equally important is the question what customer "type" really means. It is basically a set of rights or properties - defined by business. Therefore, because it belongs to the business conceptual level (domain) the "type" as a concept should *not* enter the lower levels of the portal system. What does this mean? It simply means that you should *not* see code in the portal that asks for customer "types" and derives knowledge from the type (see below: hard-coded types). The result will be portal services that are *not* specific to certain customer segmentations.

GUI

In many cases business defines a different look and feel for different customers (besides the different services available to them). During rendering of the personalized homepage, GUI classes or templates usually need to know certain things: the background color, an emotional picture etc. How does a Java Server Page get this information?

One possibility is shown is the picture below: hard-coded type information is used to derive the background color for a specific customer.

Figure 12.2.



Bad (hard-coded) Segmentation

GUI: select background color

```
type = userObject.getUserType();
If (type == LOWVALUE)
          backgroundColor="yellow";
If (type == HIGHVALUE)
          backgroundColor="purple";
```

Access Control: select service

```
type = userObject.getUserType();
If (type == LOWVALUE)
          access=NO;
If (type == HIGHVALUE)
          access=YES;
```

If the customer segmentation changes all this code needs to change!

Obviously this will lead to GUI changes in case the customer segmentation (or only the color assignments) changes.

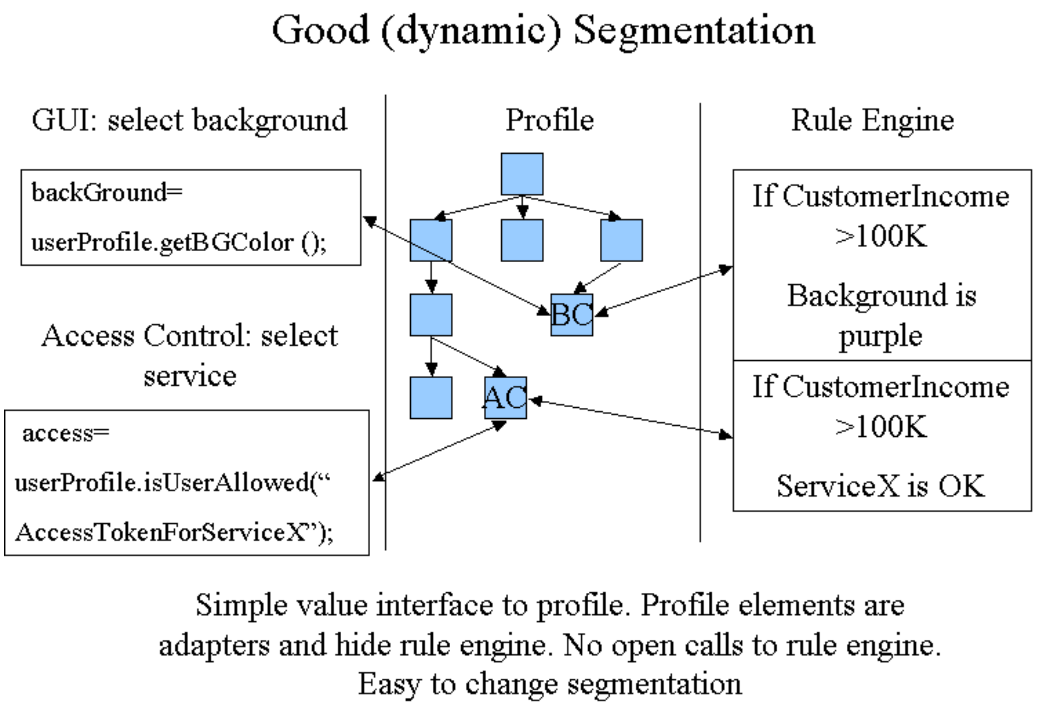
Access Control

The same goes for access control: When a homepage request enters the system, the portal needs to find out which services the customer is entitled to. It makes no sense to run services that will generate information the customer is not allowed to see. The above picture shows an implementation that uses hard-coded customer types to assign access rights. Like in the GUI case this code is fragile.

Dynamic Segmentation

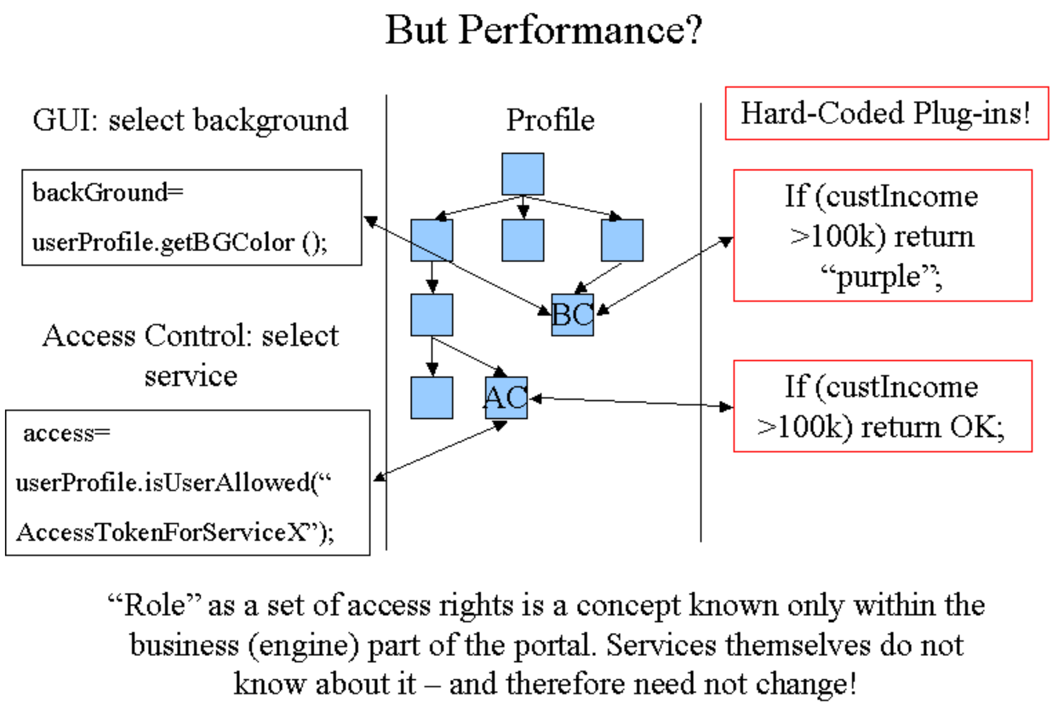
A better way to do customer segmentation is shown in the picture below:

Figure 12.3.


Good (dynamic) Segmentation
GUI: select background
backGround=
userProfile.getBGColor ();
Access Control: select service
access=
userProfile.isUserAllowed("
AccessTokenForServiceX");
Profile
BC
AC
Rule Engine
If CustomerIncome >100K
Background is purple
If CustomerIncome >100K
ServiceX is OK
Simple value interface to profile. Profile elements are adapters and hide rule engine. No open calls to rule engine. Easy to change segmentation

here the GUI asks a profile interface directly for the background color. It does not hard-code any knowledge about segmentation - in fact, it has *no concept* of customer segmentation at all. The profile interface (see below) offers a hierarchical name-value interface to clients, thereby hiding concrete implementations of the values behind its interface. Here we are using it to hide a rule engine behind the profile interface. It is obvious how the GUI code has become independent of customer type changes, but why would we want to hide the rule engine behind a profile interface? What's wrong with just calling the rule engine e.g. engine.calculateBackgroundColor(user); ? This is what we call a service interface: The rule engine works like a service and whoever needs something calls one of its service methods. The downside of such a design - as work in the CORBA services has shown -is that it is hard to change because of those service calls that get spread all over the place. And on top of it: what if we need to replace the rule engine (at least for some results) because of performance reasons? Hiding the engine behind a profile interface allows us to selectively calculate results e.g. through hard-coded plug-ins instead of going to the rule engine.

Figure 12.4.


But Performance?

```
backGround=
userProfile.getBGColor ();
```

```
If (custIncome
>100k) return
"purple";
```

```
access=
userProfile.isUserAllowed("
AccessTokenForServiceX");
```

```
If (custIncome
>100k) return OK;
```

"Role" as a set of access rights is a concept known only within the business (engine) part of the portal. Services themselves do not know about it – and therefore need not change!

The Profile Interface

The profile construct should not be confused with what many portal packages call the customer profile as a set of attributes in a customer table. Instead, it combines all kinds of customer related information and hides it behind a hierarchical name-value based tree, not unlike a DOM tree. Its biggest advantages are: It hides different data sources (e.g. some customer information coming from a Lightweight Directory Service (LDAP) base, others from the local portal database). But on top of that it allows customer information to be calculated dynamically either through a rule engine or plug-ins. The users of the profile interface will not see the difference. The result is that certain business concepts (like customer segmentation) can be restricted to the business rule level, expressed in rules for the rule engine. Changes to the business logic are changes to rules and not to portal code.

The Access Controller Interface

Roughly modeled after the J2EE security model, our portal differs in one important aspect: A user or principal may have several roles but these roles are only sets of so called access tokens. During access control (e.g. when a request needs to be validated in the controller servlet) *only the access tokens* are compared against what the service itself requires. A customers roles form the combined set of access tokens for this customer. No service understands the roles themselves. The roles belong to the business conceptual level.

This gives a lot of flexibility but requires an implementation that is both fast and maintainable. For the first release of the portal it was decided to map the access tokens to tables in the portal database. The combination of a users types into the set of access tokens was done through stored procedures that created a dynamic view of the final access tokens per customer. A propagation logic propagated new access tokens automatically to the view.

The AccessControler Interface allowed us to dynamically change the stored access tokens by asking the rule engine, much like we do in the GUI case by asking the profile interface. The AccessController Interface followed the design principle of "design for change and exception" by allowing non-hierarchical and overlapping sets of access tokens. The price to pay was an extra table mapping roles (types) to access tokens and a "non-defaults" table for those cases where individual users received special access tokens without really changing roles or getting roles added.

A very important concept that we tried to implement was "service management delegation". The ability for internal users to maintain the access rights for their parts of the portal. See: Maintainability

Rule Engine Integration

Chapter 13. Resources

References & Abbreviations

DMZ	De-Militarized Zone
DOM	Document Object Model
DOM Parser	A parser that reads a xml document an transforms it into a tree of DOM nodes
DOS Attack	Denial-Of-Service Attack
DTD	Document Type Definition
EJB	Enterprise Java Beans
J2EE	Java 2 Enterprise Edition
JSP	Java Server Pages
Thread Pool	A pool of threads used internally
QOS	Quality-Of-Service
RAS	Reliability, Availability, Stability
RMI	Remote Method Invocation
SAX Parser	A xml parser that uses the SAX event based interface to communicate docuement content
WAS	Websphere Application Server
XML	Extended Markup Language
CMS	Content Management System
EDS	External Data System

Bibliography

Most of the literature mentioned below can be found on our WIKI server under "UseCachingOnTheWeb" or "AggressiveCaching".

1. General hints and tips for web caching: UseCachingOnTheWeb
2. Information Resource Caching FAQ: http://ircache.net/Cache/FAQ
3. Dynamic Caching: A SpiderCache? flyer: www.spidercache.com
4. Class-based Cache Management for Dynamic Web Content: Discusses URL partitioning to e.g.

invalidate classes of documents

5. Cooperative Caching of Dynamic Content on a Distributed Web Server (SWALA): Discusses QOS of a distributed cache using the SWAL server

6. A Scalable and Higly Available System for Serving Dynamic Data at Frequently Accessed Web Sites (The IBM architecture for the NAGANO Olympic Site). Discusses physical architecture as well as page design for performance: put a lot on the first page)

7. JSP caching: www.jspsmart.com/

8. www.servlets.com: server site caching example from Jason Hunter. Servlet interceptor for dynamic but non-personalized page PER servlet.

9. Exploiting result equivalence in caching dynamic web content: discusses mapping of several URLs to one cached instance, e.g. mapping of map with many different locations to just a few weather forecast pages (attached)

10. A Publishing System for Efficiently Creating Dynamic Web Content, IBM Research. Shows the use of "fragments" and to speed up page creation and Object Dependency Graphs to secure and minimize update requests.

11. Oliver Vogel, Service Abstraction Layer

12. Minerva: on twiki, search for "ObjectPool"

13. Opensymphony, Oscache. A tag library for jsp caching. : http://www.opensymphony.com/oscache/ [http://www.opensymphony.com/oscache/]

14. Requirements for and Evaluation of RMI Protocols for Scientific Computing". http://www.extreme.indiana.edu/soap/sc00/paper/index.html [http://www.extreme.indiana.edu/soap/sc00/paper/index.html]

15. Design Alternatives for Scalable Web Server Accelerators (j.Song, et. Alii) IBM T.J.Watson Research Center. Uses cache arrays with CARP for caching.

16. A Middleware System Which Intelligently Caches Query Results (Degenaro et. Alii, IBM Watson). Explains the use of data update propagation (DUP) to keep caches current after database updates.

17. Fineground Condenser Product Brief. Like packeteers.com a product that does browser detection and compression. http://www.fineground.com [http://www.fineground.com]

18. Open Markets Satellite Server. A frontend that delivers dynamic content by assembling it from cached "pagelets" and stored meta-information about personalization. Used at ft.com, Europe’s largest news site. http://www.openmarket.com [http://www.openmarket.com]

19. Times Ten In-Memory Database Technology. They claim to have a ten fold performance advantage over regular RDBMS for mostly-read scenarios

20. Building and Managing Dynamic Database-Driven Web Sites. A talk from a Seybold seminar. Most important: to realize that using the typical JSP/J2EE push model (like AEPortal does) the business users wont have a chance to EDIT the dynamic sites the way they are used to e.g. with their static intranet sites). Without an information-centric "pull-model" dynamic content always implies "programmed" content.

21. Caching Dynamic Content on the World Wide Web (Jim Starz). Shows the use of chains of proxy servers and subscriptions to deliver dynamic content.

22. Einsatz von Java, XML und XSLT für grafische Oberflächen, Mike Mannion, Sven Ehrke. Uses XSLT for request processing. Originally a part of the SBC Millenium Banking (later SSP) effort). Mannion is now part of the MAP team and located in Hochstrasse 16. We need to discuss XSLT performance with him!!

23. Engineering Highly Accessed Web Sites for Performance, J.Challenger, A.Iyengar et. Alii. IBM Watson

24. Infrastructure Architecture Overview, http://columbia-test.sbcs.swissbank.com/twiki/pub/Pbit/JaDe/JADEOverview20.DOC [http://columbia-test.sbcs.swissbank.com/twiki/pub/Pbit/JaDe/JADEOverview20.DOC]

25. Graham Glass, When less is more: a compact toolkit for parsing and manipulating XML l=136,t=gr,p =electricXM http://www-106.ibm.com/developerworks/xml/library/x-elexml/index.html?open&L [http://www.mindelectric.com]

26. SSL Accelerators see: http://www.kegel.com/ssl/hw.html [http://www.kegel.com/ssl/hw.html]

27. Java Performance Tuning (Java Series (O'Reilly)) -- by Jack Shirazi;

28. Problems of double checked locking: http://www.javaworld.com/javaworld/jw-02-2001/jw-0209-toolbox_p.html [http://www.javaworld.com/javaworld/jw-02-2001/jw-0209-toolbox_p.html]

29. Http compression. www.RemoteCommunications.com/rctpd/rctpdfaq.html [http://www.RemoteCommunications.com/rctpd/rctpdfaq.html] offers both a free and a commercial version of a http compression software "proxy"

30. Speeding up the crypto: Apache e-Commerce Solutions (for ApacheCon 2000, Florida). Mark J. Cox, Geoff Thorpe. www.awe.com/mark/apcon2000 [http://www.awe.com/mark/apcon2000], www.geoffthorpe.net/geoff/apcon2000 [http://www.geoffthorpe.net/geoff/apcon2000] describes the best SSL architecture with respect to loadbalancing and performance. Suggests using a common SSL session cache and shared crypto systems.

31. http://www-4.ibm.com/software/webservers/portal/portlet.html [http://www-4.ibm.com/software/webservers/portal/portlet.html] describes what a portlet is – from different points of view (user, technical etc.)

32. Implementing a Data Cache using Readers And Writers. Billy Newport, http://www2.theserverside.com/resources/newport/ReaderWriter.html [http://www2.theserverside.com/resources/newport/ReaderWriter.html]

33. When good security leads to poor performance, Mathias Thurman, http://www.itworld.com/AppDev/1648/CWD010326STO58978/pfindex.html [http://www.itworld.com/AppDev/1648/CWD010326STO58978/pfindex.html]

34. Optimizing SSL processing for Web, Greg Govatos, http://www.itworld.com/AppDev/1684/NWW1023tech/pfindex.html [http://www.itworld.com/AppDev/1684/NWW1023tech/pfindex.html]

35. Jenny Preece, Online Communities

36. Siteminder Overview, Netegrity. http://www.netegrity.com [http://www.netegrity.com]

37. Best Practices using Http sessions, IBM white paper, http://www-106.ibm.com/developerworks/library/r-wsbest.html?n-dd-4191

[http://www-106.ibm.com/developerworks/library/r-wsbest.html?n-dd-4191] (pdf file)

38. The Future of Internet Search, Axel Uhl, Interactive Objects Software GmbH, Freiurg, Germany http://www.io-software.com [http://www.io-software.com]

39. Vignette and the J2EE Application Server, Technical White Paper http://www.vignette.com [http://www.vignette.com]

40. Schwab puts growth plan to the test, ftp://vadd1:3fzwbti2\@207.25.253.53/1/wsdd/pdf/Schwab2001.pdf [ftp://vadd1:3fzwbti2\@207.25.253.53/1/wsdd/pdf/Schwab2001.pdf]

41. CMF Dogbowl, Jeffrey P Shell's Member Page. Describes different portals and how a CMS will support them. Explains "topics". http://cmf.zope.org/CMF//Members/jshell/PortalDesigns.txt [http://cmf.zope.org/CMF//Members/jshell/PortalDesigns.txt]

42. Katherine C.Adams, Extracting Knowledge http://www.intelligentkm.com/feature/010507/feat.shtml [http://www.intelligentkm.com/feature/010507/feat.shmtl]

43. Dan Sullyvan, Beyond The Numbers http://www.intelligententerprise.com/000410/feat2.shtml [http://www.intelligententerprise.com/000410/feat2.shtml]

44. Communications of the ACM, August 2000/Vol.43 Nr. 8

45. Information Discovery, A Characterization of Data Mining Technologies and Process http://www.datamining.com/dm-tech.htm [http://www.datamining.com/dm-tech.htm]

46. Dan R.Greening, Data Mining on the Web http://www.webtechniques.com/archives/2000/01/greening.html [http://www.webtechniques.com/archives/2000/01/greening.html]

Appendix A. About the paper

This paper was written in DocBook XML [http://www.oasis-open.org/docbook/] using Emacs [http://www.gnu.org/software/emacs/], and converted to HTML using the SAXON XSLT processor from Michael Kay of ICL [http://users.iclway.co.uk/mhkay/saxon/] with a customized version of Norman Walsh's XSL stylesheets [http://www.nwalsh.com/xsl/]. From there, it was converted to PDF using HTMLDoc [http://www.easysw.com/htmldoc/], and to plain text using w3m [http://ei5nazha.yz.yamagata-u.ac.jp/~aito/w3m/eng/]

It simply copies the physical layout (entity structure) of the "Dive Into Python" book

If you're interested in learning more about DocBook for technical writing, you should read the canonical book, *DocBook: The Definitive Guide* [http://www.docbook.org/]. If you're going to do any serious writing in DocBook, I would recommend subscribing to the DocBook mailing lists [http://lists.oasis-open.org/archives/].