An Efficient Presentation-Architecture for Personalized Content

Paul Libbrecht, Carsten Ullrich, Stefan Winterstein
University of Saarland and German Research Center for Artificial Intelligence
Saarbrücken, Germany
{paul,cullrich,winter}@activemath.org

Abstract:

A possible approach for generating personalized web-sites is to compose a page out of individual elements, for instance XML-fragments, and to transform the page to the desired output format, for instance HTML. However, the required transformation processes put a heavy load on the server, hence slowing down response times significantly.

This article describes the architecture we developed to solve the performance problems that arouse out of the page generation process. In this architecture, the generation process is divided into several layers, with each layer adding/transforming well-specified data. Among other advantages, this approach allows caching of individual transformed fragments. First results confirm that in this way the performance problems can be reduced.

1 Introduction

A current trend in the WWW is the generation of personalized web pages. Personalization spans from inserting small pieces of text (such as the user's name) to composing the complete text of a page dependent on properties of the user. While early approaches were realized using CGI scripts or in-page scripts, such Java Server Pages, the advent of XML offered the possibility to directly compose a page out of XML-fragments, chosen, for instance depending on some property of the user, and to transform this page using XSLT into the desired output format (e.g. HTML).

The authors used this approach for the learning environment ACTIVEMATH [MBA+01]. In the screen-shot in Figure 1, the content fragments are clearly visible; each box correspond to a single fragment.

In the first version of ACTIVEMATH, the presentation process was realized in a somewhat "naive" matter: At every page request the presentation process was repeated completely starting from fetching the content to transforming it to the output format. This, of course, puts a heavy load on the server. Especially the transformation process requires a lot of resources.

To overcome the arising performance problems, we analyzed and structured the presentation process, and, building on this analysis, developed a general architecture for the generation of personalized web pages. In short, we divided the presentation process in

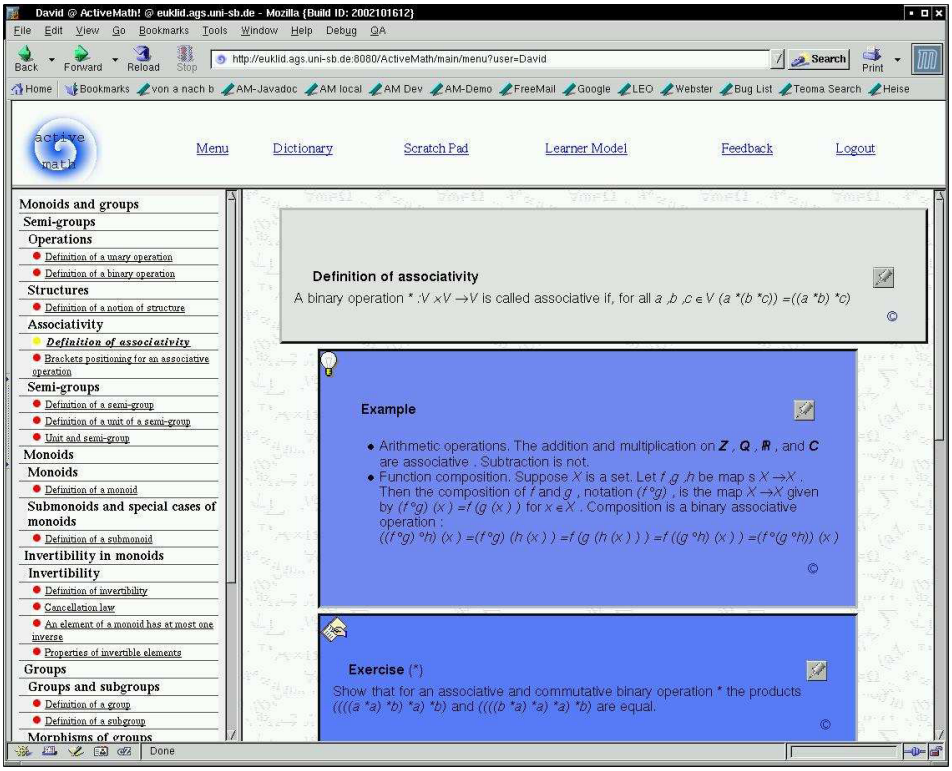

Figure 1: A screenshot of the ACTIVEMATH learning environment

several separated stages, where each stage adds distinct individual information. Thereby, caching is possible in several places.

We think this architecture is of general interest. Although developed in an educational setting, the architecture we propose is general enough to be applied in all settings that achieve personalization by choosing and composing XML-fragments.

The next section provides a detailed description the proposed architecture, followed by an preliminary analysis of the performance gains. The article concludes with a comparison of our architecture to existing frameworks.

2 Presentation Architecture

2.1 Overview

Web applications in Java can be implemented in many different ways, and for each approach, there exist numerous frameworks and support technologies, most of which are in constant flux.

For ACTIVEMATH, we chose a Model-View-Controller (MVC) architecture as the basis

for our web application. An MVC architecture separates application logic (*controller*) from application data (*model*) and the presentation of that data (*view*). It is well proven for Java web applications (see, for instance, [Jo03]), where servlets act as controllers, Java data objects form the model, and views are usually implemented by Java ServerPages (JSP), XSLT, or some other dedicated template language.

The web application part of ACTIVEMATH is based on two frameworks: MAVERICK and VELOCITY. MAVERICK [1] is a minimalist MVC framework for web publishing using Java and J2EE, focusing solely on MVC logic. It provides a wiring between URLs, Java controller classes and view templates. Further description is beyond the scope of the work.

VELOCITY [2] is a high-performance Java-based template engine, which provides a clean way to incorporate dynamic content in text based templates such as HTML pages. It provides a well focused template language with a powerful nested variable substitution and some basic control logic (`if/else`, `foreach`, `include`). VELOCITY plays an important role in ACTIVEMATH's presentation system.

2.2 Processed Data

We performed an analysis of the data that is processed/added during the presentation process of ACTIVEMATH to determine to which extent the process could be optimized with respect to the above problems. The analysis yielded the following kinds:

Content. Obviously, the presented content forms the major and most important type of data. It is expressed in the XML dialect `OMDoc` and resides in a knowledge base (e.g. an XML database). Content is mostly static in the sense that the containing text does not change (but see Section 2.4.2). However, the overall content presented to a user is of course dynamic and adapted to individual needs and learning goals.

Presentation Information. This data specifies how specific symbols (e.g. math symbols and formulas) are to be rendered for different formats such as HTML, MATHML or LaTeX. This knowledge is encoded in XSLT stylesheets.

Customization Information. While the individual learning materials that are presented to a user are covered by the above content type, additional information is added on top of the content in order to tailor the output to an individual user's needs and preferences. A good example is the indication of the state of knowledge of the user with respect to a content element. For instance, if Anton has only very limited knowledge of an element, it can be annotated in a special way, e.g. underlined with red color.

2.3 A 2-staged Presentation System

2.3.1 The Presentation Pipeline

After analyzing the presentation process of ACTIVEMATH, we chose a 2-staged approach for content generation, which uses XSLT transformations combined with a template en-

[1] MAVERICK: `http://mav.sourceforge.net/`
[2] VELOCITY: `http://jakarta.apache.org/velocity/`

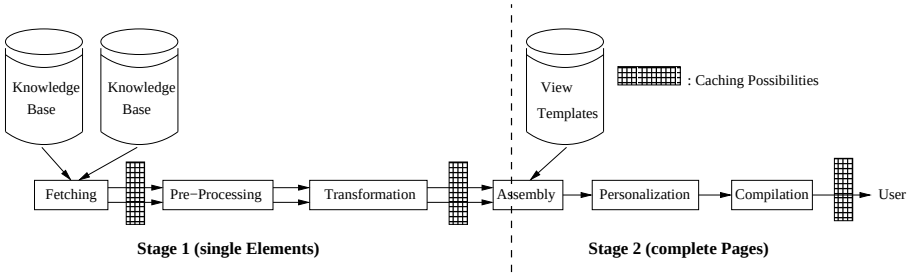

Figure 2: The Presentation Pipeline

gine (see Figure 2). Basically, the presentation pipeline can be separated into two stages: Stage 1 encompasses Fetching, Pre-Processing and Transformation, while Stage 2 consists of Assembly, Personalization and optional Compilation.

2.3.2 Stage 1: Transforming Fragments

Stage 1 deals with individual content fragments or items, which are written in OMDoc-XML and stored in a knowledge base. At this stage, content items do not depend on the user who is to view them. They have unique IDs and can be handled separately. It is only in Stage 2 that items are composed to user-specific pages.

The first part of the presentation pipeline consists of the following steps:

Fetching. Collects requested content from the knowledge base. The output of this step are XML fragments.

Pre-Processing. Inserts server-specific information into the XML content. For instance, if the content is contained in several distributed knowledge bases, this step changes the IDs of the elements to avoid duplicated IDs.

XSL Transformation. Performs the transformation into the desired output format by applying an XSLT stylesheet to the document. The output of this step are text-based content fragments in, for instance HTML or LaTeX.

Example In order to illustrate the steps of the presentation pipeline, we'll follow a simplified example. Say user Anton requests an HTML page that contains only one content item, "definition 1" with an ID of `def1`.

In a first step, this content is *fetched* from the knowledge base, which returns an OMDoc fragment:

```
<definition id="def1">
 Definition 1 with a reference to
  <ref xref="def2">definition 2</ref>
</definition>
```

The *pre-processing* step replaces the IDs of the elements. Here, it adds the name of the knowledge base as a prefix to the id:

```
<definition id="mbase://euler/def1">
 Definition 1 with a reference to
   <ref xref="mbase://euler/def2">definition 2</ref>
</definition>
```

Then the XML-fragment is *transformed* to HTML using XSLT:

```
<div class="definition" id="mbase://euler/def1">
  Definition 1 with a reference to
    $link.dict("definition 2", "mbase://euler/def2").
</div>
```

We see that the content item has been wrapped with an HTML `div` tag, and that the reference to definition 2 has been replaced by `$link.dict()`. This is actually a variable reference for the view layer and will later result in a call to a special Java helper bean, which will generate the desired HTML code for this link.

2.3.3 Stage 2: Composing Fragments

A page displayed to a user usually consists of several content items, embedded in a page template. Stage 1 delivers XSLT-transformed content fragments in the required output format. In Stage 2, these fragments are composed into a complete page and enriched with dynamic data for this specific page request.

The second half of the presentation pipeline performs of the following steps:

Assembly. Joins the fragments together to form the requested pages. The fragments in the desired output format are integrated into a page template, which is fetched from an external source.

Customization. Uses request-dependent information to add individual data to the document, such as user information, knowledge indicators or the CSS stylesheet to use.

Compilation (optional) Applies further processing to convert the generated textual content presentation into a possibly binary format like PDF (which is generated from LaTeX).

Example (continued) In the assembly step, content elements are assembled to form a complete HTML page. In our case, we only have a single content item, which is embedded in an HTML template:

```
<html> <!-- page template -->
 <head>
  <link rel="stylesheet" type="text/css" href="/css/$user.Stylesheet"/>
 </head>
 <body>
```

```
  This page is generated for $user.Name.
  <!-- begin item -->
  <div class="definition" id="mbase://euler/def1">
    Definition 1 with a reference to
    $link.dict("definition 2", "mbase://euler/def2").
  </div>
  <!-- end item -->
 </body>
</html>
```

As we see, the resulting document contains variable references (`$user.Stylesheet`, `$link.Dict`) which refer to well-defined objects available to the view layer (see next section).

In the final step, the document is interpreted by VELOCITY. All variable references are resolved and replaced by the actual text, which is then sent to the user's browser:

```
<html> <!-- page template -->
 <head>
  <link rel="stylesheet" type="text/css" href="/css/colored.css"/>
 </head>
 <body>
  This page is generated for Anton.
  <!-- begin item -->
  <div class="definition" id="mbase://euler/def1">
   Definition 1 with a reference to
   <a onClick="openInDictionary('mbase://euler/def2')">definition 2</a>.
  </div>
  <!-- end item -->
 </body>
</html>
```

2.3.4 The View Layer

As we saw in the example, the VELOCITY template engine plays a central role in Stage 2. Its major function is to replace variable references with dynamic data which is only available at request time.

This data and the names under which it is available to the view layer (i.e. VELOCITY template) is defined in our view layer interface specification. This document describes what data objects ("beans") can be accessed in each view and what properties these beans expose.

Therefore, our XSLT stylesheets can output "intermediate data" in the form of variable references, which will later be replaced by actual dynamic data.

Without the use of a template engine, dynamic data would have to be available to the XSLT stylesheet at transformation time. Among other problems (such as making the XSLT stylesheet very much dependent of the HTML layout), this would make the caching of transformed content impractical, since transformed content would be different from user to user.

Another important use of the VELOCITY view engine, its use for incremental rendering, is described in section 2.4.1.

2.3.5 Caching

Talking about optimizations only makes sense if the assumptions under which the optimizations take effect are made explicit. ACTIVEMATH was designed with two different use cases in mind:

In the group setting a large amount of users access more or less the same content. Although the specific content of the pages can differ, the materials that need to be retrieved from the knowledge base can be specified approximately. In an educational system this would correspond to learners following a lecture, where the content will be the covered domain; in an online news server this content could be the headline articles.

The second use case is the independent self-guided user. No assumptions can be taken about what content he is interested in. For this group, caching the content of one user would not help a second one, as most probably they are interested in different topics.

We designed ACTIVEMATH primarily for being used by a number of students learning about the same content. Hence, the caching we propose is mostly directed at the group case. The second case needs further investigation, although the proposed caches will not have any negative effect for the self-guided use case.

The diagram in Figure 2 indicates three points at which caching can take place:

After Fetching. ACTIVEMATH can retrieve its content from different knowledge bases that can be distributed anywhere in the web. As response times will vary heavily, caching once retrieved content at the ACTIVEMATH server can yield better performance, especially for slow connections. Caching untransformed content items also makes sense for modules that use only the content meta information, such as the title of a content element. In ACTIVEMATH, such modules are the Dictionary and the User-Model Inspector.

After Transformation. Applying XSLT-transformations is very expensive and requires a lot of resources. Therefore we decided to cache the individual elements after their transformation. The cache key is the triple *(itemId, transformationFormat, language)*.

Before delivering to the client. Another option is to cache the results of a page compilation, for instance the PDF version of a large page or book. Here, the generated file can be stored in a file-based cache on the server to avoid the expensive operation of generating again.

2.4 Additional Considerations

2.4.1 Incremental Rendering

From a user's point of view, the *perceived* performance of an application is much more important than the absolute performance measured by a stopwatch. A web application is perceived more responsive when the user can begin reading a page very quickly instead of having to wait for server to build the complete page before sending it to the browser.

For this reason, we chose an incremental rendering approach for ACTIVEMATH. Instead of sending all content through the presentation pipeline before displaying the end result to the user, the presentation pipeline is actually driven from the view layer, i.e. from the VELOCITY template. The controller logic only collects the IDs of the content items to display on a page, and – along with all other request-dependent data – passes it on to the appropriate VELOCITY template. This template has access to a special helper bean which provides a high-level function to trigger the presentation pipeline for a single content item. The helper bean will take care of fetching an item, transforming it and rendering it directly to the request's output stream.

This approach minimizes the time until the user sees the start of a new page, along with the first content. While still rendering the content further down the page, the user can already start reading the top of the page. To the user, this makes page rendering appear much faster, even if the server is not generating the page faster than it did before.

2.4.2 Randomized and Generated Content

Some content can not be cached at all because its text is generated on the fly. We distinguish between randomized and generated content:

A good example for randomized content are multiple choice exercises. Every time ACTIVEMATH presents such an exercise, the order in which the possible choices are presented is randomized. As simple as it is, it hampers students to simply copy the answers from their neighbors.

Generated content is content that is generated from an abstract representation. An example are exercises involving statistics. Statistic can be applied to a number of areas, ranging from the probability whether a person falls ill to the probability that some products sells better than others. But the underlying mathematics remain the same. Therefore, an abstract representation can specify the basics of the exercise, and the concrete instantiations are generated with respect to the field of the learner.

These kinds of dynamic content should of course not be cached after transformation in the way static content is. Instead, a more fine-grained control is needed. For instance, in the case of multiple-choice questions, transformation can occur once for each possible answer and the result can be stored in the Java object representing the exercise for a specific user. The view template can then render the exercise by accessing the exercise object directly and fetching the answer options in a random way (instead of transforming the complete exercise item and rendering it to the output stream as a whole).

So caching randomized and generated content is feasible by caching the Java object representing the instance of that content.

2.4.3 Lowering the amount of requests

As a further optimization, we also aim at avoiding the loading of whole a document for the sole purpose of performing small, local updates. For instance, changing the colored bullets in the table of content that indicate the mastery value of the user for the topics of the page can be done without requesting the page anew. Instead, these updates are performed via a JavaScript connection to the server.

This approach can be generalized to a message-based communication, making browser components *agents* communicating with the components on the server, thereby offering much more flexibility and reducing the amount of information traveling between the browser and the server.

3 Preliminary Analysis

Figure 3 includes a first theoretical estimation of the costs of the old and new architecture based on our current experiences. As we consider HTML generation, we removed the then unnecessary compilation phase. The transformation and preprocessing (which includes fetching) costs units are higher then the cost of the other stages as they take up the most resources. In the old architecture, the complete process is repeated for each request, yielding a total cost of 48 units. With the restructured presentation process and the then used caches, only a very limited number of request make it to the transformation stage, thereby requiring an average cost of 4.3, about a tenth of the old cost.

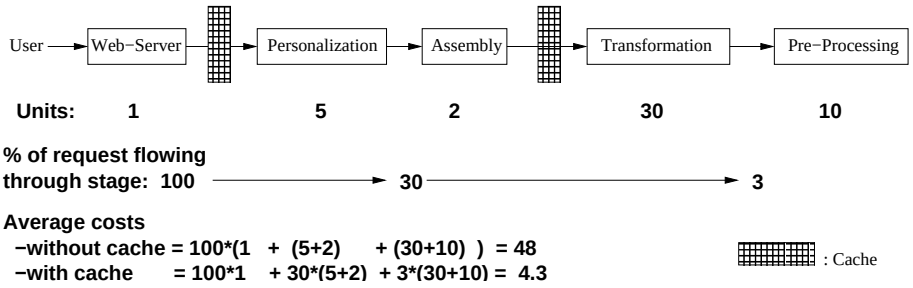

Figure 3: Theoretical estimation

First practical results support this estimation. We compared the times taken up for page presentation of un-cached elements and cached elements. The number of elements on a page ranged from two to ten. Without a cache, the required time lay between 1600 and 6100 milliseconds, with the caches between 80 and 210 milliseconds, a result that greatly exceeded our expectations. It seems that the required transformation costs is even higher than estimated.

4 Related Work and Conclusion

A huge quantity of systems in education offer individualization of content, see, for instance, [Br99]. There, Brusilovsky provides a comprehensive overview on adaptive hypermedia techniques, to a great extent, however, focusing on manipulating pre-made pages, for example hiding/showing a paragraph of text, enabling/disabling hyperlinks, or changing the sequence in which the pages are presented to the user. Recent systems that provide the possibility of assembling pages and courses from smaller *learning objects* depending on user properties are [CWBG02] or [SKK+02]. Sadly, they do not provide details regarding the technical aspects focused in this article.

Cocoon[3] is a web development framework based heavily on XML and XML transformation pipelines. In contrast to Cocoon, we believe that XML documents are not an ideal basis for data objects and that XML and related technologies should not spread throughout our entire application. Instead, we try to restrict the usage of XML to the edges of our application wherever possible, relying more on the traditional mechanisms for program logic and data that the underlying programming system (Java in our case) is offering us.

To summarize, we proposed an efficient layered presentation architecture that can be applied in all settings where single XML-fragments serve as the basis of personalization. We obtain the following benefits by following a 2-staged approach with a hybrid XSLT and template engine approach:

- XSLT-transformed content items are independent of user data. This allows for effective caching, since content is personalized only late in the presentation pipeline.
- XSLT stylesheets are not dependent of the view layout. Instead, they focus on transforming content into different formats.
- view layout (e.g. HTML) is only contained in template files, where it is easily changeable even by non-programmers, such as web designers. Even more, the whole look&feel of our application can be exchanged by just switching to another set of templates.

References

[Br99] Brusilovsky, P.: Adaptive and intelligent technologies for web-based education. *Künstliche Intelligenz*. 4:19–25. 1999.

[CWBG02] Conlan, O., Wade, V., Bruen, C., und Gargan, M.: Multi-model, metadata driven approach to adaptive hypermedia services for personalized elearning. In: Bra, P. D., Brusilovsky, P., und Conejo, R. (Hrsg.), *Proceedings of the second International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems*. volume 2347 of *LNCS*. S. 100–111. Springer-Verlag. 2002.

[Jo03] Johnson, R.: *Expert One-on-One J2EE Design and Development*. Wrox. 2003.

[MBA+01] Melis, E., Büdenbender, J., Andres, E., Frischauf, A., Goguadze, G., Libbrecht, P., Pollet, M., und Ullrich, C.: Activemath: A generic and adaptive web-based learning environment. *Artificial Intelligence and Education*. 12(4). 2001.

[SKK+02] Specht, M., Kravcik, M., Klemke, R., Pesin, L., und Hüttenhain, R.: Adaptive learning environment for teaching and learning in WINDS. In: Bra, P. D., Brusilovsky, P., und Conejo, R. (Hrsg.), *Proceedings of the second International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems*. volume 2347 of *LNCS*. S. 572–575. Springer-Verlag. 2002.

[3] Cocoon: `http://cocoon.apache.org/`